Avaliação de Ferramentas de Processamento de Linguagem Natural para Verificação da Sintaxe de Modelos de Requisitos

Arthur H. M. de Oliveira[1], Francisco Kleber R. Castro[1], Johnny C. Marques[1], Christopher S. Cerqueira[1]

[1]Instituto Tecnológico de Aeronáutica (ITA) CEP 12228-900 – São José dos Campos – SP – Brazil

hendricks@ita.br,franciscofkrc@ita.br,johnny@ita.br,chris@ita.br

Abstract. *This article addresses the problem of imprecision and ambiguity inherent in software requirements formulated in natural language, a challenge that frequently leads to erroneous interpretations, rework, and failures in development projects. The search for automated methods that ensure the syntactic conformity of these requirements to industry-adopted models has been widely explored in the literature. In this context, aiming to investigate the feasibility and performance of such automation processes, this study evaluated Natural Language Processing (NLP) tools in Python – Stanza, NLTK, and SpaCy – for the syntactic verification of requirements based on the Rupp template. The results indicate that the NLTK tool excelled in both precision and processing efficiency, offering a promising solution to assist automated syntactic verification algorithms.*

Keywords: Systems Engineering (SE), Requirements Engineering (RE), Requirements Templates (RTs), Natural Language Processing (NLP)

Resumo. *Este artigo aborda a problemática da imprecisão e ambiguidade inerente aos requisitos de software formulados em linguagem natural, um desafio que frequentemente acarreta interpretações errôneas, retrabalho e falhas em projetos de desenvolvimento. A busca por métodos automatizados que garantam a conformidade sintática desses requisitos a modelos adotados pela indústria tem sido amplamente explorada na literatura. Neste contexto, buscando investigar a viabilidade e o desempenho de tais processos de automação, este estudo avaliou as ferramentas de Processamento de Linguagem Natural (PLN) em Python – Stanza, NLTK e SpaCy – na verificação sintática de requisitos baseados no modelo Rupp. Os resultados indicam que a ferramenta NLTK se destacou em precisão e eficiência de processamento, oferecendo uma promissora solução para auxiliar algoritmos de verificação sintática automatizada de requisitos.*

Palavras-chave: Engenharia de Sistemas (ES), Engenharia de Requisitos (ER), Modelos de Requisitos, Processamento de Linguagem Natural (PLN)

Área Temática: *Sistemas Complexos, Pensamento Sistêmico e Teorias Sistêmicas*

1. Introdução

A transformação digital tem revolucionado paradigmas no desenvolvimento de *software*, proporcionando uma agilidade inédita através da automação de processos, o que resulta na redução dos ciclos de desenvolvimento e entrega, além de ampliar a capacidade de inovação ao longo do ciclo de vida dos produtos (GRUHN; STRIEMER, 2018). Esses pilares tornaram a adaptação ágil dos sistemas uma vantagem competitiva crucial, permitindo que as empresas atendam rapidamente às necessidades de negócios e reduzam o tempo de respostas às novas tendências em um mercado extremamente dinâmico.

Nesse contexto, a Engenharia de Requisitos (ER) emerge como uma disciplina essencial na fase de desenvolvimento de *software*, visando a produção de sistemas com qualidade (ALMENDRA, 2022). Por se tratar de um processo intensivo em conhecimento, os processos de ER partem de dados dispersos em múltiplas fontes – como as partes interessadas, regulamentações, produtos já existentes e a experiência da equipe, entre outros, e resultam em especificações estruturadas e decisões identificadas, compreendidas e negociadas entre as partes interessadas (MARQUES; CUNHA, 2019).

Esse processo abrange atividades cruciais como a elicitação, especificação, modelagem e validação dos requisitos, fundamentais para assegurar a qualidade do *software* (NECULA; DUMITRIU; GREAVU-ȘERBAN, 2024). Essas atividades frequentemente demandam intervenção humana, sendo a Linguagem Natural a notação predominante para expressar requisitos na prática industrial (ZHAO *et al.*, 2021). Contudo, requisitos formulados em linguagem natural irrestrita são intrinsecamente imprecisos e suscetíveis a ambiguidades (ARORA; GRUNDY; ABDELRAZEK, 2015). Essa inerente imprecisão e o potencial para múltiplas interpretações podem gerar problemas significativos que comprometem o desenvolvimento do sistema, caso não sejam endereçados em fases iniciais (BALWANI et al., 2024).

A fim de mitigar esses problemas, os Modelos de Requisitos têm sido amplamente adotados pela indústria para a modelagem de *software* (BALWANI *et al.*, 2024). Tais modelos consistem em estruturas predefinidas que especificam e documentam requisitos, organizando-os em campos específicos para a captura de informações. Eles impõem uma estrutura sintática fixa, o que ajuda a mitigar a ambiguidade. Exemplos incluem o *Easy Approach to Requirements Syntax* (EARS) proposto por MAVIN *et al.* (2009) e o modelo Rupp proposto por POHL; RUPP (2021). Atrelado a isso, o Processamento de Linguagem Natural (PLN) emerge como uma solução promissora para automatizar a verificação de conformidade sintática, aproveitando a prevalência da linguagem natural nos requisitos (YALLA; SHARMA, 2015). Abordagens automatizadas empregam *pipelines* de PLN em conjunto com regras para analisar e validar a sintaxe dos requisitos de forma eficiente (JÚNIOR *et al.,* 2024) (BARBOSA *et al.,* 2024) (OLIVEIRA *et al.,* 2025). Algoritmos como AutoReco (BALWANI *et al.*, 2024), RUBRIC (ARORA *et al.*, 2014) e RETA (ARORA *et al.*, 2015) já demonstram a aplicação dessas técnicas para a verificação de conformidade e diagnóstico automático, superando limitações de algoritmos que dependem de ontologias como DODT (FARFELEDER *et al.*, 2011). Apesar dos recentes avanços, a literatura existente destaca que a maioria das *pipelines* de PLN empregadas nesses estudos utilizam predominantemente a ferramenta GATE (*General Architecture for Text Engineering*), com a verificação de padrões realizada via JAPE (*Java Annotations Pattern Engine*). Essa dependência aponta para uma lacuna e um considerável potencial de investigação em relação

a outras ferramentas e técnicas de PLN. Assim, a exploração de alternativas, especialmente aquelas desenvolvidas em linguagens de programação como Python, pode não apenas aprimorar a eficiência dos processos de verificação, mas também oferecer maior flexibilidade e adaptabilidade às necessidades específicas da análise de requisitos.

Diante disso, este artigo visa avaliar ferramentas de Processamento de Linguagem Natural (PLN) para a verificação automatizada da sintaxe de Modelos de Requisitos, em linguagem Python, examinando métricas de precisão e eficiência.

2. Processamento de Linguagem Natural para Engenharia de Requisitos

O papel importante da Linguagem Natural na Engenharia de Requisitos (ER) tem sido estabelecido há muito tempo (ZHAO *et al.*, 2021). A evidência empírica ao longo dos anos demonstrou que a Linguagem Natural é a notação mais comum para expressar requisitos na prática industrial, com 95% dos casos no início dos anos 2000 e 61% mais recentemente (NECULA; DUMITRIU; GREAVU-ȘERBAN, 2024). Nesse contexto, o Processamento de Linguagem Natural (PLN), um componente crucial da Inteligência Artificial (IA), surge como uma ponte entre a linguagem humana e a linguagem de máquina, permitindo que as máquinas processem, analisem e compreendam dados de linguagem humana, além de gerar respostas semelhantes às humanas (NECULA; DUMITRIU; GREAVU-ȘERBAN, 2024).

A aplicação do PLN na Engenharia de Requisitos, ou o termo do inglês *Natural Language Processing for Requirements Engineering* (NLP4RE), é uma área de pesquisa e desenvolvimento que busca aplicar tecnologias (técnicas, ferramentas e recursos) de PLN ao processo de ER para auxiliar analistas humanos em diversas tarefas de análise linguística em documentos de requisitos textuais (ZHAO *et al.*, 2021). De acordo com ZHAO *et al.* (2021), esses três tipos de tecnologias se diferenciam, podendo ser amplamente categorizados em:

- Técnicas de Processamento de Linguagem Natural: métodos, processos ou procedimentos para executar uma tarefa particular de PNL.
- Ferramentas de Processamento de Linguagem Natural: sistemas de *software* ou bibliotecas que oferecem suporte a uma ou mais técnicas de PNL.
- Recursos de Processamento de Linguagem Natural: dados linguísticos que dão suporte a técnicas ou ferramentas de PNL, que pode ser um léxico de linguagem ou um *corpus* (coleção de textos).

2.1. Técnicas de Processamento de Linguagem Natural

O Mapeamento Sistemático da Literatura (MSL) de NLP4RE realizado por ZHAO *et al.* (2021) revela que as técnicas de PNL comumente utilizadas em tarefas de ER incluem:

- Marcação de Parte da Fala (do inglês *Part-of-Speech tagging* ou ***POS tagging***): processo de atribuir uma categoria gramatical a cada palavra em um texto.
- Tokenização (do inglês ***Tokenization***): dividir um texto em unidades menores e significativas, chamadas "*tokens*".
- Análise Sintática (do inglês ***Parsing***): processo de analisar a estrutura gramatical de uma frase para determinar suas relações sintáticas e hierárquicas.
- Remoção de palavras de parada (do inglês ***Stop-words removal***): palavras muito comuns em um idioma que geralmente não carregam um significado semântico significativo por si só, como artigos, preposições, conjunções e pronomes.

- Extração de Termos (do inglês ***Term extraction***): identificar automaticamente termos relevantes, que podem ser palavras únicas ou expressões multi-palavras, em coleções de textos especializados ou de domínio específico.
- Radicalização (do inglês ***Stemming***): técnica de normalização de texto que reduz as palavras flexionadas (variantes de uma palavra) à sua forma radical ou base.

2.2. Ferramentas de Processamento de Linguagem Natural

ZHAO *et al.* (2021) também revela as ferramentas de PLN de uso geral que suportam essas técnicas e que são frequentemente utilizadas em pesquisas de NLP4RE:

- **Stanford CoreNLP**: conjunto abrangente de ferramentas de PLN desenvolvido pela Universidade de Stanford (MANNING *et al.*, 2014).
- **GATE** (*General Architecture for Text Engineering*): *framework* de código aberto para desenvolvimento e implantação de *software* de PLN (CUNNINGHAM, 2002).
- **NLTK** (*Natural Language Toolkit*): biblioteca de código aberto popular para Python, amplamente utilizada para ensino e pesquisa em PLN (BIRD, 2006).
- **Apache OpenNLP**: kit de ferramentas de PLN baseado em Java para processamento de texto em linguagem natural.
- **WEKA** (*Waikato Environment for Knowledge Analysis*): conjunto de ferramentas de *software* de aprendizado de máquina escrito em Java, desenvolvido na Universidade de Waikato, Nova Zelândia (GARNER, 1995).

2.3. Recursos de Processamento de Linguagem Natural

Segundo ZHAO *et al.* (2021) os recursos de PNL mais usados para NLP4RE são:

- **WordNet**: banco de dados lexical no idioma inglês.
- **VerbNet**: biblioteca de classes de verbos computacionais no idioma inglês.
- **British National Corpus**: coleção de textos escritos e falados da língua inglesa britânica contemporânea.

A escolha pela avaliação das ferramentas de PLN neste trabalho é, principalmente, devido a inerente capacidade dessas ferramentas integrarem e aplicarem uma vasta gama de recursos e técnicas de PLN em seus *pipelines* e bibliotecas.

3. Modelos de Requisitos

Diversos Modelos de Requisitos foram propostos na literatura de Engenharia de Requisitos, e adotados na prática industrial (ARORA *et al.*, 2015). A importância destes modelos reside em sua capacidade de padronizar a forma como os requisitos são escritos, o que promove maior clareza, reduz ambiguidades e facilita a verificação e o processamento automatizado. Exemplos notáveis incluem o modelo de Rupp (POHL; RUPP, 2021) e o *Easy Approach to Requirements Syntax* (EARS) (MAVIN *et al.*, 2009). Embora a abordagem apresentada neste trabalho possa ser adaptada para trabalhar com uma variedade de modelos existentes, optou-se por focar no modelo Rupp, dado o seu vasto uso na indústria como aponta ARORA *et al.* (2015).

3.1. Modelo Rupp

O modelo Rupp, proposto por POHL; RUPP (2021), estabelece que a formulação de um requisito pode ser composta por até seis características distintas: uma condição opcional, o

nome do sistema, um verbo modal, a funcionalidade de processamento necessária (dividida em três tipos distintos de requisitos), o objeto para o qual a funcionalidade é necessária e detalhes opcionais, como mostra a Figura 1.

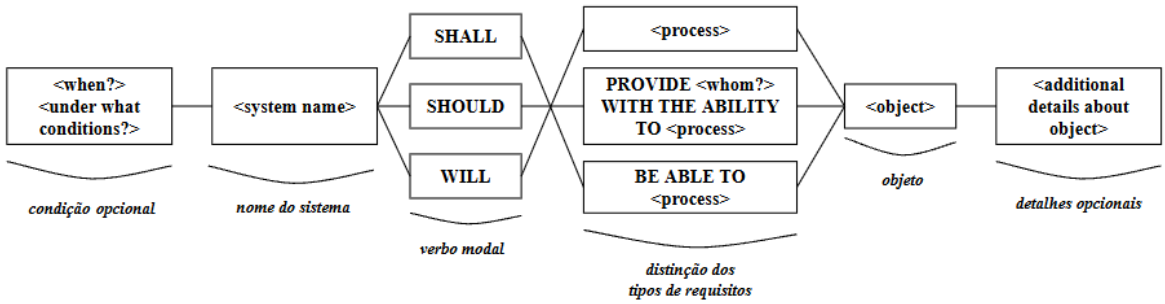

Figura 1. Modelo Rupp.

Segundo ARORA *et al.* (2015), o modelo Rupp categoriza a funcionalidade de processamento em três tipos distintos de requisitos:

- **Requisitos Autônomos**: descrevem a funcionalidade que o sistema executa independentemente da interação do usuário. Utilizam o formato <*process*> (onde <*process*> representa a funcionalidade em questão).
- **Requisitos de Interação com o Usuário**: referem-se à funcionalidade que o sistema oferece a usuários específicos. O formato é *"PROVIDE <whom?> WITH THE ABILITY TO <process>"* (sendo o usuário <*whom?*> e a funcionalidade <*process*>).
- **Requisitos de Interface**: definem a funcionalidade que o sistema realiza em resposta a eventos acionados por outros sistemas. O formato é "*BE ABLE TO <process>*" (onde <*process*> descreve a capacidade de processamento do sistema em reação a um evento externo).

4. Metodologia

A metodologia utilizada neste trabalho para a verificação de conformidade sintática de modelos de requisitos a partir de ferramentas de PLN se baseou na proposta de ARORA *et al.* (2015) para o algoritmo RETA, que descreve um fluxograma das etapas do processo com um *pipeline* de PNL. Nesse sentido, o fluxograma proposto da Figura 2 apresenta as etapas desse processo, que conta com um *pipeline* de PNL com dois módulos distintos: um Tokenizador (que divide o texto em palavras e símbolos) e um Marcador de Parte da Fala (que atribui *tags* de classes gramaticais). Os requisitos em Linguagem Natural compõem os dados de entrada, e a saída é o resultado da verificação de conformidade sintática.

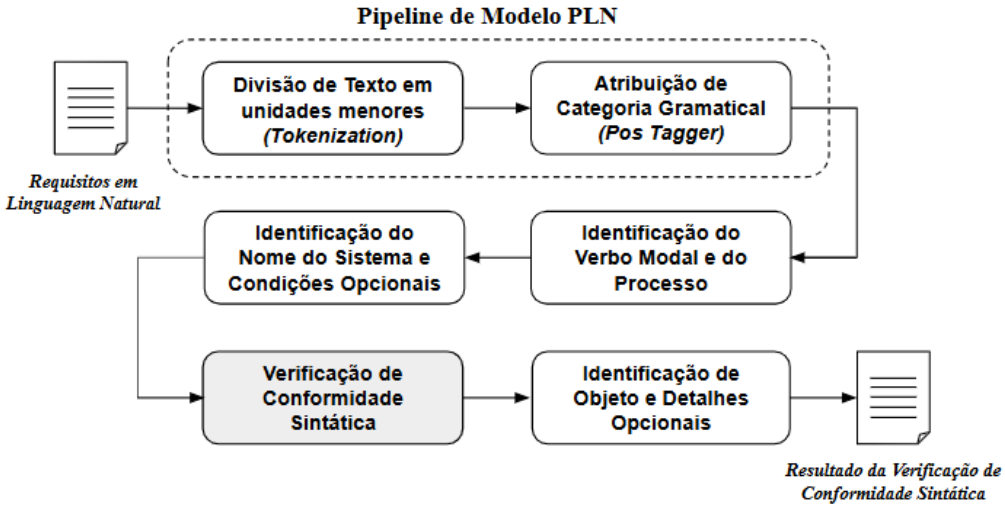

Figura 2. Fluxograma de Verificação da Sintaxe de Modelos de Requisitos utilizando Processamento de Linguagem Natural.

4.1. Modelos PLN

Este trabalho se concentrou em ferramentas de Processamento de Linguagem Natural (PLN) desenvolvidas para a linguagem Python, dada a sua popularidade e disponibilidade de bibliotecas. Outro critério utilizado para essa decisão foi a predominância de ferramentas na literatura focadas apenas na utilização de GATE (*General Architecture for Text Engineering*) e JAPE (*Java Annotations Pattern Engine*). Para tal, foram selecionadas a versão Python do Stanford CoreNLP, conhecida como Stanza, NLTK e SpaCy.

- **Stanza**: é um pacote Python, também desenvolvido pelo Stanford Natural Language Processing Group (QI *et al.*, 2020), que oferece acesso fácil a modelos de última geração para uma ampla gama de tarefas de PLN, incluindo tokenização, marcação de classes gramaticais e lematização (*https://stanfordnlp.github.io/stanza/*).
- **NLTK** (*Natural Language Toolkit*): incorpora interfaces para mais de 50 *corpora* e recursos lexicais, como o WordNet, e técnicas como tokenização, *POS Tagging*, dentre outras (BIRD, 2006) (*https://www.nltk.org/index.html*).
- **SpaCy**: é uma biblioteca de código aberto de modelos pré-treinados que foram desenvolvidos usando grandes conjuntos de dados e algoritmos avançados de aprendizado de máquina, capazes de entender e processar a linguagem humana em uma extensão significativa, também contando com técnicas como tokenização e *POS Tagging* (*https://spacy.io/*).

4.2. Divisão de Texto em Unidades Menores (*Tokenization*)

A primeira atividade do *pipeline* de PLN é dividir o texto em "*tokens*". Para isso, considere o requisito exemplo:

"For each communication channel type, the system SHALL BE ABLE TO maintain a configurable timeout parameter."

O resultado esperado dessa operação é um *array* com os respectivos *tokens*:

["For", "each", "communication", "channel", "type", ",", "the", "system", "SHALL", "BE", "ABLE", "TO", "maintain", "a", "configurable", "timeout", "parameter", "."]

4.3. Atribuição de Categoria Gramatical (*Pos Tagger*)

A segunda atividade do *pipeline* de PLN consiste em elencar os atributos gramaticais aos *tokens*. Considere o mesmo requisito utilizado anteriormente com a atribuição de categoria gramatical feita pela biblioteca SpaCy:

ADP	DET	NOUN	NOUN	NOUN	PUNCT	DET	NOUN	AUX
For	each	communication	channel	type	,	the	system	SHALL

AUX	NOUN	PART	VERB	DET	DET	NOUN	NOUN	PUNCT
BE	ABLE	TO	maintain	a	configurable	timeout	parameter	.

Cada palavra ou *token* na composição da frase tem sua respectiva classe gramatical, como o "*maintain*" que é classificado como "*VERB*", representando o processamento do sistema no requisito, assim como o "*SHALL*" que é o verbo modal auxiliar do requisito, e apresenta a *tag* "*AUX*". Os padrões das *tags* podem divergir dependendo da ferramenta em utilização. SpaCy e Stanza, por exemplo, seguem o padrão *Universal Dependencies* (UD) para suas *tags* de Parte da Fala. Dessa forma, suas *tags* são iguais, enquanto NLTK segue o padrão *Penn Treebank Tagset*. A Tabela 1 apresenta as principais classes gramaticais utilizadas neste projeto com as suas respectivas *tags* por ferramenta de PLN.

Tabela 1. Classes Gramaticais por Ferramenta de PLN.

Classe Gramatical	SpaCy	NLTK	Stanza
Verbo	*VERB*	*VB (forma base)* *VBD (pretérito)* *VBG (particípio presente ou gerúndio)* *VBN (particípio passado)* *VBP (presente)* *VBZ (presente, 3ª pessoa do singular)*	*VERB*
Pontuação	*PUNCT*	*. , :*	*PUNCT*
Preposições	*ADP*	*IN*	*ADP*
Conjunção Subordinativa	*SCONJ*	*IN*	*SCONJ*
Determinante	*DET*	*DT*	*DET*

4.4. Identificação do Verbo Modal e do Processo

A Tabela 2 apresenta o pseudocódigo utilizado para extrair o Verbo Modal e o Processo nos requisitos em Linguagem Natural.

Tabela 2. Pseudocódigo para Identificação do Verbo Modal e do Processo.

```
1.  def extract_modal_verb_process(tokens, upos_tags):
2.
3.    modals = ["shall", "should", "will"]
4.    found_modals = []
5.    found_process = []
6.    modal_check = False
7.
8.    for i, token in enumerate(tokens):
9.
10.     if modal_check == False:
11.       if token in modals:
12.         found_modals.append(token)
13.         modal_check = True
14.
15.     else:
16.       if modal_check == True
17.         # Muda para NLTK de acordo com Tabela 1
18.         if upos_tags[i] == “VERB”:
19.           found_process.append(token)
20.         break
21.
22.   return found_modals, found_process
```

A função “*extract_modal_verb_process*” da Linha 1 recebe os *tokens* e a respectiva classe gramatical através da variável “*upos_tags*” em formato de *array*. A Linha 10 verifica se o *token* é um verbo modal, ao comparar com a lista da Linha 3, dentre as opções “*shall*”, “*should*” e “*will*” do modelo Rupp. Após a identificação do verbo modal, a Linha 18 verifica se o próximo *token* é um verbo através da comparação com a *string* “*VERB*” (esse valor muda para o NLTK de acordo com a Tabela 1). Esse pseudocódigo foi resumido para propósito de síntese, mas outras funções de recomendação como ausência de verbo modal ou processo, e até quando o requisito apresenta mais de um verbo modal são parte da complexidade dessa verificação no código original. Ao final, a função retorna as variáveis “*found_modals*” e “*found_process*” com o(s) modal(s) encontrado(s) e o processo.

4.5. Identificação do Nome do Sistema e Condições Opcionais

A Tabela 3 apresenta o pseudocódigo utilizado para extrair o Nome do Sistema e Condições Opcionais nos requisitos em Linguagem Natural.

Tabela 3. Pseudocódigo para Identificação do Nome do Sistema e Condições Opcionais.

```
1.  def extract_system_name_optional_conditions(tokens, upos_tags):
2.
3.    modals = ["shall", "should", "will"]
4.    tokens_upside_down = []
5.    duty_anchor = []
6.    optional_conditions = []
7.    system_name = []
8.    modal_check = False
9.    PP_check = False
10.
11.   tokens_upside_down = tokens[::-1]
```

```
12.
13.  for token in tokens_upside_down:
14.
15.    if modal_check == False:
16.      if token in modals:
17.        modal_check = True
18.
19.    else:
20.      if modal_check == True:
21.        duty_anchor.append(token)
22.
23.  for i, token in enumerate(tokens):
24.
25.    tokens_anchor_upside_down = []
26.    upos_tags_anchor_upside_down = []
27.
28.    tokens_anchor_upside_down.append(token)
29.    upos_tags_anchor_upside_down.append(upos_tags[i])
30.
31.    tokens_anchor_upside_down = tokens_anchor_upside_down[:len(duty_anchor)]
32.    tokens_anchor_upside_down= tokens_anchor_upside_down[::-1]
33.    tokens_anchor_upside_down= list(map(str, tokens_anchor_upside_down))
34.
35.    upos_tags_anchor_upside_down = upos_tags_anchor_upside_down[:len(duty_anchor)]
36.    upos_tags_anchor_upside_down = upos_tags_anchor_upside_down[::-1]
37.    upos_tags_anchor_upside_down = list(map(str, upos_tags_anchor_upside_down))
38.
39.  for token_item in range(len(tokens_anchor_upside_down)):
40.
41.     if (PP_check == True):
42.
43.          optional_conditions.append(tokens_anchor_upside_down[token_item])
44.
45.     else:
46.
47.          if (upos_tags_anchor_upside_down[token_item] == "PUNCT" or
48.              upos_tags_anchor_upside_down[token_item] == "SCONJ" or
49.              upos_tags_anchor_upside_down[token_item] == "VERB"):
50.              PP_check = True
51.              optional_conditions.append(text_test_upside_down[token_item])
52.
53.          elif (upos_tags_anchor_upside_down[token_item] == "DET"):
54.              PP_check = True
55.              system_name.append(tokens_anchor_upside_down[token_item])
56.
57.          else:
58.              system_name.append(tokens_anchor_upside_down[token_item])
59.
60.  system_name = system_name[::-1]
61.  optional_conditions= optional_conditions[::-1]
62.
63.  return system_name, optional_conditions
```

A função "*extract_system_name_optional_conditions*" da Linha 1 também recebe os *tokens* e a respectiva classe gramatical através da variável "*upos_tags*" em formato de *array*. A Linha 11 organiza esse *array* de trás para frente. A Linha 16 verifica se o *token* é um verbo

modal, ao comparar com a lista da Linha 3, dentre as opções "*shall*", "*should*" e "*will*" do modelo Rupp. Após a identificação do verbo modal, a Linha 20 começa a captura do nome do sistema e das condições opcionais através da variável "*duty_anchor*". As Linhas 41 e 47 verificam se alguma pontuação, preposição, conjunção subordinativa ou determinante foram encontradas. Se sim, as condições opcionais começam a ser coletadas através da variável "*optional_conditions*" na Linha 43 e 51. Se não, o nome do sistema começa a ser capturado através da variável "*system_name*" na Linha 55 e 58. Esse pseudocódigo também foi resumido para propósito de síntese, porém outras funções de recomendação como ausência de verbo modal ou processo, múltiplos verbos modais no requisito, ausência de nome do sistema fazem parte da complexidade dessa verificação no código original, gerando recomendações de erro. Ao final do código, a função retorna as variáveis "*system_name*" e "*optional_conditions*" com o nome do sistema e condições opcionais encontradas.

4.6. Verificação de Conformidade Sintática

A Tabela 4 apresenta o pseudocódigo utilizado para Classificar o Tipo de Requisito a partir do processo encontrado nas etapas anteriores, utilizando a variável "*found_process*".

Tabela 4. Pseudocódigo para Classificação de Tipo de Requisito.

```
1.  def verify_template(found_process):
2.
3.    template = ""
4.
5.    if found_process == "provide":
6.
7.      template = 'Interaction Requirements'
8.
9.    elif found_process == "be":
10.
11.     template = 'Interface Requirements'
12.
13. else:
14.
15.     template = 'Autonomous Requirements'

      return template
```

As Tabelas 5, 6 e 7 apresentam os pseudocódigos utilizados para a Verificação de Conformidade Sintática do modelo Rupp para Requisitos Autônomos, Requisitos de Interação com o Usuário e Requisitos de Interface.

Tabela 5. Pseudocódigo para Verificação de Conformidade Sintática de Requisitos Autônomos

```
1.  def verify_autonomous_requirements(found_process, tokens, upos_tags):
2.
3.    verification = "Modelo incorreto."
4.
5.    for i, token in enumerate(tokens):
6.
7.      if token == found_process:
8.
```

```
9.      # Muda para NLTK de acordo com Tabela 1
10.     if upos_tags[i] == “VERB”:
11.       found_process.append(token)
12.       verification = "Modelo correto."
13.       break
14.
15.  return verification
```

A função “*verify_autonomous_requirements*” da Linha 1 recebe a variável processo, os *tokens* e a respectiva classe gramatical através da variável “*upos_tags*” em formato de *array*. A Linha 7 verifica se o *token* é o próprio processo. Se for, a Linha 10 verifica se essa palavra é um verbo, a partir de verificação da classe gramatical “*VERB*” (esse valor muda para o NLTK de acordo com a Tabela 1). Esses requisitos categorizam o modelo Rupp para requisitos autônomos. Ao final, a função retorna a informação se o requisito segue o modelo Rupp para Requisitos Autônomos.

Tabela 6. Pseudocódigo para Verificação de Conformidade Sintática de Requisitos de Interação com o Usuário.

```
1.  def verify_interaction_requirements(tokens, upos_tags, found_modals, system_name,
2.  optional_condition):
3.
4.    whom = []
5.    found_process = []
6.    verification = "Modelo incorreto."
7.    message = ""
8.    whom = ""
9.
10.   for i in range(len(optional_condition + system_name + found_modals)):
11.
12.       tokens.pop(0)
13.   phrase_check = "with the ability to "
14.
15.   indice = string_tokens_cleanned.find(phrase_check)
16.
17.  if indice == -1:
18.     message = "Interaction Requirement Template requires a 'WITH THE ABILITY TO'
19.                       keyword before the process"
20.
21.   elif indice != -1:
22.     whom = string_tokens_cleanned[:indice]
23.
24.   if(len(whom) == 0):
25.
26.       message = "Requirement has no whom"
27.
28.   process = string_tokens_cleanned[indice + len(phrase_check):]
29.
30.   process = tokens_cleanned[0]
31.
32.   if (len(process) == 0):
33.
34.       message = "Requirement has no process"
35.
36.   if (message = ""):
```

```
37.
38.     message = "Modelo correto."
39.
40.  return verification
```

A função "*verify_interaction_requirements*" da Linha 1 recebe os *tokens*, a respectiva classe gramatical através da variável "*upos_tags*", o verbo modal através da variável "*found_modals*", o nome do sistema através da variável "*system_name*", as condições opcionais através da variável "*optional_condition*", em formato de *array*. A Linha 12 remove as condições opcionais, o nome do sistema, o verbo modal para o início de verificação da sintaxe do requisito. A Linha 13 verifica se o requisito apresenta a frase "*with the ability to* ", que é mandatória no tipo de requisito de interação com o usuário. Em caso positivo, é identificado o sujeito que interage com o sistema através da palavra "*whom*" nas Linhas 21 e 22, e o processo através da palavra "*process*" nas Linhas 28 e 30. Esses requisitos categorizam o modelo Rupp para requisitos de interação com o usuário. Para propósito de síntese, esse pseudocódigo apresenta apenas a captura das variáveis, mas outras funções de recomendação como ausência da palavra "*whom*" ou das palavras chave "*with the ability to* " fazem parte da complexidade dessa verificação no código original, gerando recomendações de erro. Ao final, a função retorna a informação se o requisito segue o modelo Rupp para Interação com o Usuário.

Tabela 7. Pseudocódigo para Verificação de Conformidade Sintática de Requisitos de Interface.

```
1.  def verify_interface_requirements(tokens, upos_tags, found_modals, system_name,
2.  optional_condition):
3.
4.    keywords = ["able", "to"]
5.    found_process = []
6.    verification = "Modelo incorreto."
7.    message = ""
8.
9.    for i in range(len(keywords)):
10.
11.       if i == 0 and tokens[i] != "able":
12.
13.         message = "Interface Requirement Template requires a 'BE ABLE TO' keyword before
14.                    the process"
15.         break
16.
17.       if i == 1 and tokens[i] != "to":
18.
19.         verify_template = "Template 3 Non-Conformance"
20.         message = "Interface Requirement Template requires a 'BE ABLE TO' keyword before
21.                    the process"
22.
23.   process = tokens_cleanned[0]
24.
25.   if (len(process) == 0):
26.
27.       message = "Requirement has no process"
28.
29.   if (message = ""):
```

```
30.
31.     message = "Modelo correto."
32.
33.  return verification
```

A função "*verify_interface_requirements*" da Linha 1 recebe os *tokens*, a respectiva classe gramatical através da variável "*upos_tags*", o verbo modal através da variável "*found_modals*", o nome do sistema através da variável "*system_name*", as condições opcionais através da variável "*optional_condition*", em formato de *array*. A Linha 12 remove as condições opcionais, o nome do sistema, o verbo modal para início de verificação da sintaxe do requisito. A Linha 11 e 17 verifica se as próximas palavras do requisito após o verbo "*be*" são "*able*" e "*to*", mandatórias no tipo de requisito de interface. Em caso positivo, o processo é identificado através da palavra "*process*" na Linha 23. Esses requisitos categorizam o modelo Rupp para requisitos de interface. Para propósito de síntese, esse pseudocódigo apresenta apenas a captura das variáveis, mas outras funções de recomendação como ausência das palavras chave "*be*", "*able*" ou "*to*" fazem parte da complexidade dessa verificação no código original, gerando recomendações de erro. Ao final, a função retorna a informação se o requisito segue o modelo Rupp para Interface.

4.7. Identificação de Objeto e Detalhes Opcionais

A Tabela 8 apresenta o pseudocódigo utilizado para extrair o Objeto e Detalhes Opcionais nos requisitos em Linguagem Natural.

Tabela 8. Pseudocódigo para Identificação de Objeto e Detalhes Opcionais.

```
1.  def extract_object_details(tokens, upos_tags, found_modals, found_process, system_name,
2.  optional_condition, template):
3.
4.    verification = "Modelo incorreto."
5.    object = []
6.    details = []
7.    PP_check = False
8.
9.    for i in range(len(optional_condition + system_name + found_modals + found_process)):
10.
11.       tokens.pop(0)
12.
13.    if template == 'Interaction Requirements':
14.      for i in range(len(6)):
15.       tokens.pop(0)
16.
17.   elif template == 'Interface Requirements'
18.       for i in range(len(3)):
19.         tokens.pop(0)
20.
21.   for i, token in enumerate(tokens):
22.
23.       if PP_check == False:
24.
25.           if (upos_tags[i] == "VERB" or upos_tags[i] == "PUNCT"):
26.              PP_check = True
27.
```

```
28.         else:
29.             object.append(token)
30.
31.     else:
32.         details.append(token)
33.
34.  return object, details
```

A função "*extract_object_details*" da Linha 1 recebe os *tokens*, a respectiva classe gramatical através da variável "*upos_tags*", o verbo modal através da variável "*found_modals*", o nome do sistema através da variável "*system_name*", as condições opcionais através da variável "*optional_condition*", o processo através da variável "*found_process*", em formato de *array*. A Linha 11 remove as condições opcionais, o nome do sistema, o verbo modal, e o processo. Em seguida, dependendo do tipo de requisito, as palavras-chave serão eliminadas nas Linhas 15 e 19. A Linha 25 identifica se o *token* é um verbo ou uma pontuação, a partir da verificação das classes gramaticais "*VERB*" e "*PUNCT*" (esses valores mudam para o NLTK de acordo com a Tabela 1). Em caso positivo, a coleta de dados relacionados ao objeto vai para detalhes. Para propósito de síntese, esse pseudocódigo apresenta apenas a captura das variáveis, mas outras funções de recomendação como ausência do objeto fazem parte da complexidade dessa verificação no código original, gerando recomendações de erro. Ao final, a função retorna o objeto e os detalhes opcionais a partir das variáveis "*object*" e "*details*".

5. Experimentos

Os experimentos foram realizados considerando o modelo RUPP e as três bibliotecas de PLN que utilizam Python: Stanza, NLTK e SpaCy.

5.1. Base de Dados de Requisitos

Foi utilizada a base de dados de requisitos de *software* da plataforma Kaggle chamada "*Software requirements dataset*" que pode ser acessada no site da plataforma: *https://www.kaggle.com/datasets/iamvaibhav100/software-requirements-dataset*, na data 4 de Julho de 2025.

Este *dataset* é composto por 977 entradas, excluindo os cabeçalhos, e apresenta duas colunas principais: "*Type*" e "*Requirement*", onde a coluna "*Requirement*" descreve textualmente os requisitos e a coluna "*Type*" o categoriza. A classificação dos requisitos no *dataset* segue uma taxonomia detalhada, que distingue entre os duas categorias principais: Requisitos Funcionais ("*F*" ou "*FR*"), que especificam o que o sistema deve fazer, e Requisitos Não Funcionais ("*RNF*"), que definem como o sistema deve operar, incluindo atributos de qualidade e restrições. O *dataset* também traz outras categorias de requisitos não funcionais, utilizando as seguintes classificações: Disponibilidade (A), Tolerância a Falhas (FT), Legal (L), Aparência (LF), Manutenibilidade (MN), Operacional (O), Desempenho (PE), Portabilidade (PO), Escalabilidade (SC), Segurança (SE) e Usabilidade (US). A Figura 3 apresenta a quantidade de requisitos por categoria.

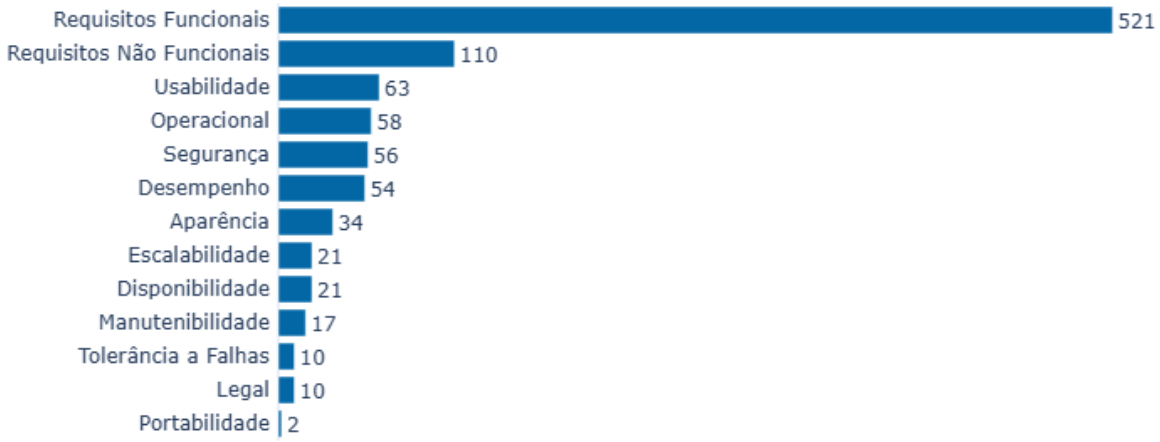

Figura 3. Quantidade de Requisitos por Categoria.

Este projeto utilizou apenas os 521 Requisitos Funcionais.

5.2. Ambiente de Execução dos Códigos em Python

Foi utilizada a ferramenta Google Colab (*https://colab.google*) para a execução dos códigos em linguagem Python. Essa ferramenta consiste em uma plataforma de Jupyter notebooks hospedada na nuvem que simplifica significativamente o desenvolvimento e a execução de projetos de aprendizado de máquina e ciência de dados. Ela oferece acesso gratuito a recursos computacionais robustos, como Unidades de Processamento Gráfico (GPUs) e Unidades de Processamento Tensorial (TPUs).

5.3. Métricos de Avaliação

Neste projeto, o desempenho do modelo foi avaliado utilizando um conjunto de métricas-chave, começando pelos conceitos fundamentais de **Verdadeiros Positivos (VP)**, que representam as instâncias corretamente identificadas como positivas, e **Verdadeiros Negativos (VN)**, que são as instâncias corretamente identificadas como negativas. Complementarmente, foram considerados os **Falsos Positivos (FP)**, que ocorrem quando uma instância negativa é erroneamente classificada como positiva, e os **Falsos Negativos (FN)**, que se referem a instâncias positivas classificadas incorretamente como negativas.

A partir desses valores, foram calculadas métricas mais abrangentes: a **Acurácia**, que indica a proporção de previsões corretas (VP + VN) em relação ao total de previsões; a **Precisão**, que mede a proporção de verdadeiros positivos entre todas as previsões positivas (VP / (VP + FP)), sendo crucial em cenários onde falsos positivos são custosos; o **Recall** (ou Sensibilidade), que avalia a proporção de verdadeiros positivos entre todas as instâncias positivas reais (VP / (VP + FN)), importante quando falsos negativos devem ser minimizados; e a **Pontuação F1** (ou F1-Score), que é a média harmônica da Precisão e do Recall, oferecendo um equilíbrio entre as duas e sendo particularmente útil em conjuntos de dados desbalanceados.

Além dessas métricas de desempenho, o **Tempo de Execução** foi considerado, avaliando a eficiência computacional do modelo - vital para a otimização de recursos.

6. Resultados e Discussões

Os resultados obtidos foram divididos em seções de análise do tipo de sugestão gerada e análise dos métricos considerados no trabalho.

6.1. Análise de Sugestões

A Figura 4 apresenta a distribuição da quantidade de requisitos por tipo de requisito do modelo Rupp. Em outras palavras, essa distribuição avalia a classificação dos tipos de requisitos pelas ferramentas de PLN.

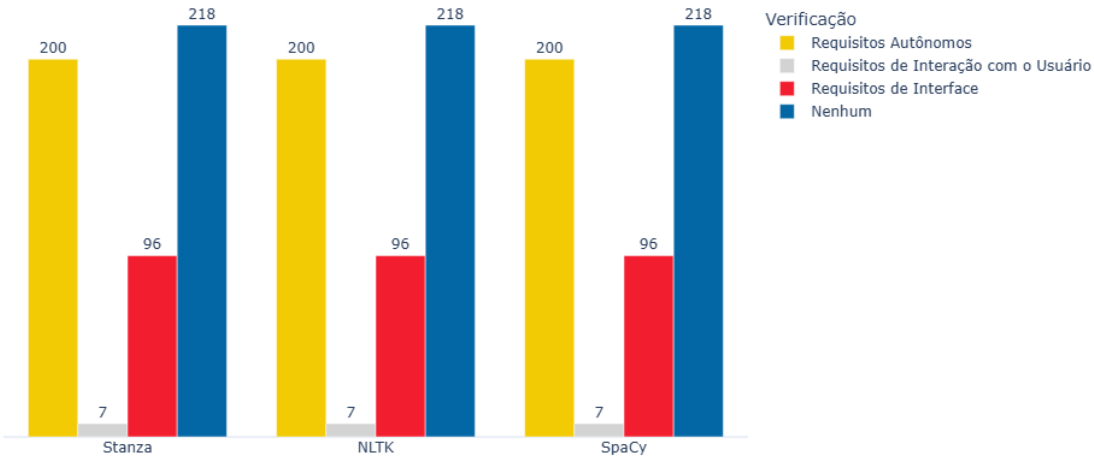

Figura 4. Resultado da Classificação dos Tipos de Requisitos do Modelo Rupp por ferramenta PLN.

O resultado apresentado na Figura 4 mostra que todas as três ferramentas (Stanza, NLTK e SpaCy) classificaram os requisitos igualmente. Isso se deve ao fato de que o pseudocódigo da Tabela 4 utilizou apenas a função de tokenização e comparação de *strings* para a classificação dos tipos de requisitos do modelo Rupp. Dessa forma, não há mudanças significativas nessa funcionalidade para as diferentes ferramentas utilizadas.

A Figura 5 apresenta as recomendações comuns entre os três tipos de requisitos do modelo Rupp, incluindo a verificação dos pseudocódigos das Tabelas 2, 3, 5, 6, 7 e 8. Percebe-se que as classes "*Falta Verbo Modal*" e "*Múltiplos Verbos Modais*" apresentam os mesmos 182 requisitos e 36 requisitos, respectivamente, por ferramenta. Esses valores foram computados utilizando o pseudocódigo da Tabela 2, e apesar da Linha 18 do pseudocódigo utilizar a *tag* de classe gramatical da Parte de Fala que difere entre as ferramentas - Stanza e a SpaCy utilizando o padrão *Universal Dependencies* (UD) e a NLTK utilizando o padrão *Penn Treebank Tagset,* não houveram mudanças significativas entre as ferramentas PLN. Além disso, nenhuma das ferramentas apresentou sugestões em relação à falta de nome do sistema ou falta de processo, utilizando o pseudocódigo da Tabela 3.

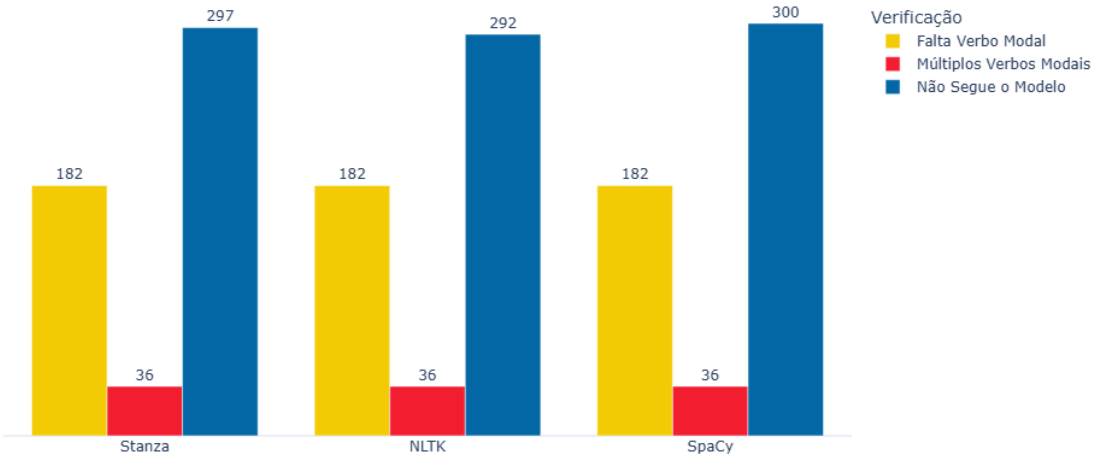

Figura 5. Resultado das Recomendações Comuns entre os Tipos de Requisitos do Modelo Rupp por ferramenta PLN.

No que tange à verificação "*Não Segue o Modelo*", observa-se uma diferenciação no desempenho das ferramentas. Vale ressaltar que esse dado reflete também informações de "*Falta de Verbo Modal*" e "*Múltiplos Verbos Modais*", uma vez que a falta ou excesso desses elementos caracteriza uma quebra de padronização do modelo Rupp. A ferramenta Stanza, por exemplo, identificou 297 requisitos que não aderem ao modelo Rupp. Desses, 182 não apresentavam verbo modal e 36 continham múltiplos verbos modais. Assim, 79 dos requisitos não conformes do Stanza foram exclusivamente decorrentes de funções específicas associadas ao tipo de requisito em análise. De maneira similar, as ferramentas NLTK e SpaCy registraram, respectivamente, 292 e 300 requisitos que não seguiam o modelo Rupp. Dentre esses, 74 (NLTK) e 82 (SpaCy) apresentam problemas de não conformidade atribuídos às particularidades de funções específicas do tipo de requisito.

Observando os resultados específicos das bibliotecas Stanza e SpaCy, 79 (Stanza) e 82 (SpaCy), nota-se uma curiosidade em relação à diferença dos seus resultados, uma vez que ambas utilizarem o padrão *Universal Dependencies* (UD) para a classificação de *tags* de Parte de Fala. Isso sugere que, apesar de possíveis semelhanças relacionadas à UD, as bases de dados de treinamento utilizadas nessas ferramentas podem ser diferentes, fazendo com que as estruturas lexicais sejam interpretadas e classificadas de forma distinta. Por outro lado, a variação no resultado da biblioteca NLTK (74) pode ser explicada tanto pelo mesmo fator de diferentes bases de dados como pelo uso do padrão *Penn Treebank Tagset*, que difere as classes gramaticais do UD, como visto na Tabela 1.

Dessa forma, a fim de aprofundar a compreensão dessas distinções, a Figura 5 detalha as recomendações específicas para os três tipos de requisitos do modelo Rupp, complementadas pela verificação dos pseudocódigos nas Tabelas 5, 6, 7 e 8. Em outras palavras, essa figura apresenta uma setorização da categoria "*Não Segue o Modelo*" da Figura 4 em recomendações específicas dos tipos de requisitos do modelo Rupp.

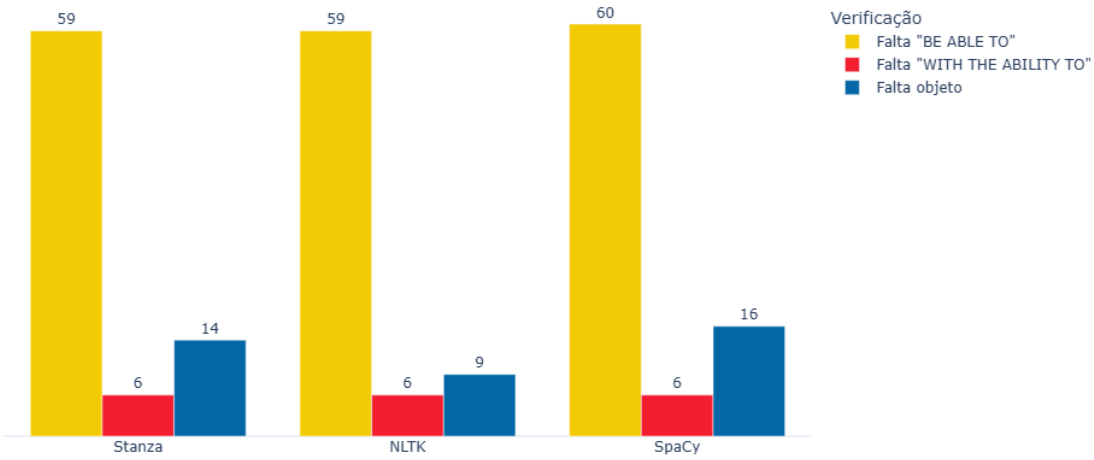

Figura 5. Resultado das Recomendações Específicas entre os Tipos de Requisitos do Modelo Rupp por ferramenta PLN.

Ao avaliar a Figura 5, notou-se que a ferramenta SpaCy apresentou 60 problemas com o modelo Rupp para Requisitos de Interface através da verificação "*Falta 'BE ABLE TO*", enquanto que as bibliotecas Stanza e NLTK apresentaram 59.

Essa diferença de um requisito avaliado como problema pela biblioteca SpaCy foi investigada, e foi encontrado o seguinte requisito com esse problema específico:

" shall be able to stream purchased movie via Windows Media Player for up to 2 days aftere purchase date."

Esse requisito apresenta um espaço no início da frase, antes da palavra "*shall*", e esse detalhe foi capturado durante a Tokenização e categorização de Parte de Fala:

[" ", "shall", "be", "able", "to", "stream", "purchased", "movie", "via", "Windows", "Media", "Player", "for", "up", "to", "2", "days", "aftere", "purchase", "date", "date"]

SPACE	AUX	AUX	ADJ	PART	VERB	VERB	NOUN	ADP	PROPN
	shall	be	able	to	stream	purchased	movie	via	Windows

NOUN	NOUN	ADP	ADP	ADP	NUM	NOUN	ADP	NOUN	NOUN	PUNCT
Media	Player	for	up	to	2	days	aftere	purchase	date	.

Por outro lado, as bibliotecas Stanza e NLTK desconsideram esse espaço nos processos de Tokenização e categorização de Parte de Fala, antecipando esse potencial erro de digitação.

As barras vermelhas na Figura 5 permaneceram iguais para as três ferramentas, com 6 problemas com o modelo Rupp para Requisitos de Interação com o Usuário através da verificação "*Falta 'WITH THE ABILITY TO'*". Entretanto, esse dado pode ser explicado pelo

fato de que existem apenas 7 requisitos categorizados como Interação com o Usuário, de acordo com a Figura 4. Em um cenário com uma quantidade maior de requisitos, pode haver uma diferença considerável, como o exemplo anterior da classificação do espaço entre as palavras como parte do corpo do requisito.

As maiores diferenças se concentram na verificação "*Falta Objeto*", com 14 (Stanza) 9 (NLTK) e 16 (SpaCy). Para tal, uma das principais causas está no fato de que o código considera as *tags* de verbo como um mecanismo de parada para começar a coleta dos detalhes opcionais. Assim, considere o seguinte requisito:

"The estimator shall apply selected recycled parts to the collision estimate."

Com a seguinte classificação de *tags* gramaticais:

DT	**NN**	**MD**	**VB**	**VBN**	**JJ**	**NNS**	**TO**	**DT**
The	estimator	shall	apply	selected	recycled	parts	to	the

NN	**NN**	**.**
collision	estimate	.

O objeto "*selected recycled parts*" apresenta "*selected*" com *tag* "*VBN*" que, de acordo com a Tabela 1, é um verbo, e mecanismo de parada para o pseudocódigo da Tabela 8. Como esse exemplo, outros requisitos foram identificados com a mesma situação. Assim, 6 (Stanza), 4 (NLTK) e 4 (SpaCy) representam oportunidades de melhoria de lógica do código, cerca de 1.15%, 0.77% e 0.77%, respectivamente, do total de 521 requisitos.

A quantidade de itens verificados corretamente pelas bibliotecas foram 7 (Stanza), 5 (NLTK) e 5 (SpaCy). E a quantidade de itens verificados incorretamente foram 5 (Stanza), 0 (NLTK) e 1 (SpaCy). Dentre os erros, a biblioteca SpaCy apresentou o mesmo problema de reconhecer o espaço em branco durante a Tokenização e categorização de Parte de Fala, reconhecendo o processo como um elemento em branco, o que resultou em uma pausa no mecanismo de coleta do objeto como o caso do requisito:

"System should be able to display "No results found" in case no search results found."

Com a seguinte classificação de *tags* gramaticais:

NOUN	**AUX**	**AUX**	**ADJ**	**PART**	**SPACE**	**VERB**	**PUNCT**	**DET**	**NOUN**
System	should	be	able	to		display	"	No	results

VERB	**PUNCT**	**ADP**	**NOUN**	**DET**	**SPACE**	**NOUN**	**NOUN**	**VERB**	**PUNCT**
found	"	in	case	no		search	results	found	.

Outro erro ocorreu com requisitos que apresentavam pontuação como aspas após o processo, como o caso de:

"In case no image available , application should return "no image" flag."

Com a seguinte classificação de *tags* gramaticais:

ADP	NOUN	DET	NOUN	ADJ	PUNCT	NOUN	AUX	VERB
In	case	no	image	available	,	application	should	return

PUNCT	PUNCT	DET	PUNCT	NOUN	PUNCT
“	no	image	“	flag	.

A única biblioteca que não apresentou esse erro foi a NLTK devido ao fato dos caracteres de pontuação precisarem ser especificados. Para Stanza e SpaCy esses caracteres estão contidos dentro da classe "*PUNCT*", causando falsos alertas.

6.2. Análise de Métricos

A Tabela 9 apresenta a Acurácia, Precisão, Recall e Pontuação F1 dos experimentos.

Tabela 9. Acurácia, Precisão, Recall e Pontuação F1 dos Experimentos.

	Ferramenta		
Métrica	**Stanza**	**NLTK**	**SpaCy**
Verdadeiros Positivos (VP)	234	234	234
Verdadeiros Negativos (VN)	224	229	221
Falsos Positivos (FP)	63	58	66
Falsos Negativos (FN)	0	0	0
Acurácia	0.8791	0.8887	0.8733
Precisão	0.7879	0.8014	0.7800
Recall	1.0000	1.0000	1.0000
Pontuação F1	0.8814	0.8897	0.8764

A partir da Tabela 9, pode-se observar que a ferramenta NLTK apresentou a melhor acurácia, precisão e pontuação *F1* comparada às ferramentas Stanza e SpaCy. Esse resultado ressalta a análise anterior, e também a particularidade da biblioteca no uso do padrão *Penn Treebank Tagset*. Pode-se observar também que a métrica *recall* foi igual para todas as

ferramentas (1) que avalia a proporção de verdadeiros positivos entre todas as instâncias positivas reais (VP / (VP + FN)), importante quando falsos negativos devem ser minimizados.

A Tabela 10 apresenta o tempo de execução em segundos de cada ferramenta para o total de 521 requisitos. A biblioteca NLTK apresentou o melhor resultado com 0.5129 segundos, comparada a biblioteca SpaCy com 5.7852 segundos. A biblioteca Stanza apresentou o pior desempenho em tempo de execução com cerca de 629.8088 segundos.

Tabela 10. Tempo de Execução dos Experimentos.

	Ferramenta		
Métrica	**Stanza**	**NLTK**	**SpaCy**
Tempo de Execução (s)	629.8088	0.5129	5.7852

7. Conclusão e Trabalhos Futuros

A ambiguidade inerente aos requisitos de *software* formulados em linguagem natural representa uma problemática persistente e crítica na Engenharia de Requisitos, frequentemente culminando em erros, retrabalhos dispendiosos e atrasos significativos ao longo do ciclo de desenvolvimento. Para mitigar esses desafios e aprimorar a qualidade dos requisitos, a verificação sintática automatizada surge como uma solução promissora. Neste contexto, este artigo avaliou a precisão e a eficiência das ferramentas de Processamento de Linguagem Natural (PLN) – Stanza, NLTK e SpaCy, desenvolvidas em Python – para a verificação automatizada da sintaxe em modelos de requisitos, utilizando como base o modelo Rupp. Os resultados obtidos demonstraram que a ferramenta NLTK se destacou em critérios de precisão como acurácia, precisão e pontuação *F1*, assim como em critério de eficiência de tempo de execução. Dessa forma, esta pesquisa busca contribuir para o campo da Engenharia de Requisitos ao oferecer *insights* práticos sobre a aplicação de diferentes ferramentas de PLN para melhorar a qualidade dos requisitos, reduzindo a ambiguidade e o potencial de erros em fases iniciais do ciclo de desenvolvimento de *software*.

Para trabalhos futuros, recomenda-se ampliar o escopo da análise, incluindo outros modelos de requisitos, como o modelo EARS, a fim de verificar a generalização dos resultados obtidos. Adicionalmente, a investigação com outras bases de dados de requisitos, que possivelmente sejam maiores e mais diversificadas, poderá enriquecer significativamente a validade externa do estudo. Um campo particularmente promissor reside na exploração de técnicas avançadas de Processamento de Linguagem Natural (PLN), incluindo a aplicação de *Large Language Models* (LLMs). Essa abordagem não se limitaria apenas à verificação sintática, mas também buscaria explorar a capacidade de correção automatizada de requisitos que não aderem aos padrões estabelecidos pelos modelos, o que aprimoraria ainda mais o processo de garantia de qualidade em Engenharia de Requisitos.

8. Referências

ALMENDRA, Camilo et al. How assurance case development and requirements engineering interplay: a study with practitioners. Requirements Engineering, v. 27, p. 273–292, 2022. Springer.

ARORA, Chetan et al. Requirement boilerplates: Transition from manually-enforced to automatically-verifiable natural language patterns. In: 2*014 IEEE 4th International Workshop on Requirements Patterns (RePa)*. IEEE, 2014. p. 1-8.

ARORA, Chetan et al. Automated checking of conformance to requirements templates using natural language processing. *IEEE transactions on Software Engineering*, v. 41, n. 10, p. 944-968, 2015.

ARORA, Chetan; GRUNDY, John; ABDELRAZEK, Mohamed. Advancing requirements engineering through generative ai: Assessing the role of llms. *In: Generative AI for Effective Software Development*. Cham: Springer Nature Switzerland, 2024. p. 129-148.

BALWANI, Shivani et al. An Approach for Providing Recommendation for Requirements Non-Conformant with Requirement Templates (RTs). In: Proceedings of the 17th Innovations in Software Engineering Conference. 2024. p. 1-11.

BARBOSA, Leonardo de Mello et al. Mean dependency length—a new metric for requirements quality. In: *INCOSE International Symposium*. 2024. p. 1021-1035.

BIRD, Steven. NLTK: the natural language toolkit. In: *Proceedings of the COLING/ACL 2006 interactive presentation sessions.* 2006. p. 69-72.

CUNNINGHAM, Hamish. GATE, a general architecture for text engineering. *Computers and the Humanities*, v. 36, p. 223-254, 2002.

OLIVEIRA, Arthur Hendricks Mendes, et al. Model-Driven Systems Engineering Automation with Artificial Intelligence for Robustly Writing Automotive System Requirements. No. 2025-01-8658. SAE Technical Paper, 2025.

FARFELEDER, Stefan et al. DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development. In: *14th IEEE international symposium on design and diagnostics of electronic circuits and systems*. IEEE, 2011. p. 271-274.

GARNER, Stephen R. et al. Weka: The waikato environment for knowledge analysis. In: *Proceedings of the New Zealand computer science research students conference.* 1995. p. 57-64.

GRUHN, Volker; STRIEMER, Rüdiger. *The Essence of Software Engineering*. Springer Nature, 2018.

JÚNIOR, Fernando Sarracini et al. Systems engineering process enhancement: Requirements verification methodology using natural language processing (NLP) for automotive industry. SAE Technical Paper, 2024.

KHAN, Javed Ali; QAYYUM, Shamaila; DAR, Hafsa Shareef. Large Language Model for Requirements Engineering: A Systematic Literature Review. 2025.

MANNING, Christopher D. et al. The Stanford CoreNLP natural language processing toolkit. In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations.* 2014. p. 55-60.

MARQUES, J.; CUNHA, A. M. da. A set of requirements for certification of airborne military software. In: IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), 2019, San Diego, CA, USA. IEEE, 2019. p. 1-7.

MAVIN, Alistair et al. Easy approach to requirements syntax (EARS). In: *2009 17th IEEE international requirements engineering conference*. IEEE, 2009. p. 317-322.

NECULA, Sabina-Cristiana; DUMITRIU, Florin; GREAVU-ȘERBAN, Valerică. A systematic literature review on using natural language processing in software requirements engineering. *Electronics*, v. 13, n. 11, p. 2055, 2024.

POHL, Klaus; RUPP, Chris. B*asiswissen requirements engineering: Aus-und Weiterbildung nach IREB-Standard zum certified professional for requirements engineering foundation level*. dpunkt. verlag, 2021.

QI, Peng et al. Stanza: A Python natural language processing toolkit for many human languages. *arXiv preprint arXiv:2003.07082*, 2020.

YALLA, Prasanth; SHARMA, Nakul. Integrating natural language processing and software engineering. *International Journal of Software Engineering and Its Applications*, v. 9, n. 11, p. 127-136, 2015.

ZHAO, Liping et al. Natural language processing for requirements engineering: A systematic mapping study. *ACM Computing Surveys (CSUR)*, v. 54, n. 3, p. 1-41, 2021.