III Escola Regional de Informática do Piauí. Livro Anais - Artigos e Minicursos, v. 1, n. 1, p. 336-360, jun, 2017.
www.eripi.com.br/2017 - ISBN: 978-85-7669-395-6

Capítulo
5

Introdução ao Processamento de Linguagem Natural usando Python

Jardeson Leandro Nascimento Barbosa, João Paulo Albuquerque Vieira,
Roney Lira de Sales Santos, Gilvan Veras Magalhães Junior,
Mariana dos Santos Muniz, Raimundo Santos Moura

Abstract

The main objective of this short course is to present an introduction to Natural Language Processing (NLP) using Python's Natural Language Toolkit (NLTK). This course also includes practical examples and basic applications that will enable its participants to practice the knowledge acquired during the presentation. This course is suitable for any student versed in basic programming skills or linguistics. It is expected that by the end of it, the participants will be able to understand key aspects of NLP and to perform essential NLP pre-processing tasks.

Resumo

O objetivo principal deste minicurso é apresentar uma introdução ao Processamento de Linguagem Natural (PLN) usando a ferramenta Natural Language Toolkit (NLTK) do Python. Este minicurso também inclui exemplos práticos e aplicações básicas que permitirão que os participantes pratiquem os conhecimentos adquiridos durante a apresentação. Este curso é adequado para estudantes versados em habilidades básicas de programação ou linguística. Espera-se que ao final do curso, os participantes sejam capazes de entender aspectos-chave do PLN e de realizar tarefas de pré-processamento essenciais de PLN.

5.1. Introdução

O Processamento de Linguagem Natural (PLN) é uma área da computação que tem como objetivo extrair representações e significados mais completos de textos livres escritos em linguagem natural [Indurkhya and Damerau, 2010]. Refere-se por "linguagem natural" uma linguagem que é usada para comunicações do dia-a-dia feitas por humanos; línguas

como Português, Inglês ou Mandarim. Em contraste às linguagens artificiais, como linguagens de programação e notações matemáticas, as linguagens naturais têm evoluído à medida que passam de geração para geração, e são difíceis de definir com regras explícitas [Bird et al., 2009].

A grosso modo, o PLN pode ser definido como uma forma de descobrir quem fez o quê, a quem, quando, onde, como e porquê [Robertson, 1946]. O PLN geralmente faz uso de conceitos linguísticos como classes de palavras (substantivo, verbo, adjetivo, etc.), aqui chamadas de *Part-Of-Speech*, além de estruturas gramaticais. O PLN também lida com situações mais complexas, como anáforas e ambiguidades. Isso se dá através de várias representações de conhecimento, como léxicos de palavras e seus significados; propriedades e regras gramaticais da linguagem; tesauros de sinônimos ou abreviações; e ontologias de entidades e ações [Indurkhya and Damerau, 2010].

Além de tratar o texto como uma mera sequência de caracteres, o PLN considera a estrutura hierárquica da linguagem. Ao analisar a linguagem pelo seu significado, os sistemas de PLN podem ser utilizados em uma diversa gama de papéis, como corretores gramaticais, na conversão de fala para texto, na tradução automática de texto entre linguagens e na análise de sentimentos/mineração de opiniões [Liu and Zhang, 2013].

Como a linguagem humana é raramente escrita ou falada corretamente, entender a linguagem humana é entender não apenas as palavras, mas os conceitos e como eles estão ligados entre si para criarem significados. Apesar da facilidade que temos, como humanos, de aprender a linguagem natural, a ambiguidade de uma linguagem é o que torna o PLN um problema difícil para computadores.

O PLN é importante por razões científicas, econômicas, sociais e culturais e está experimentando um rápido crescimento à medida que suas teorias e métodos são implantados em uma variedade de novas tecnologias de linguagem. Por esta razão, é importante para uma vasta gama de pessoas ter um conhecimento dos trabalhos da área. Dentro da indústria, isso inclui profissionais na interação humano-computador, análise de informações de negócios e desenvolvimento de *software*. Dentro da academia, inclui pesquisadores em áreas de linguística, ciência da computação e inteligência artificial [Bird et al., 2009]. Por esses motivos, é importante expor aspectos teóricos e práticos como forma de um minicurso abordando conceitos clássicos de Processamento de Linguagem Natural e Mineração de Texto à estudantes, pesquisadores e profissionais de subáreas ligadas à Tecnologia da Informação e Linguística.

A estrutura final do minicurso será organizada da seguinte forma: a Seção 2 apresentará conceitos-chave de PLN e de linguística, além de discutir o processo de mineração de texto; na Seção 3 serão apresentados o Python e o NLTK, bem como estruturas de dados e métodos básicos para o pré-processamento de PLN; na Seção 4 serão discutidos problemas práticos de PLN e será demonstrado como escrever programas simples que ajudam a manipular e analisar dados de linguagem utilizando o NLTK; finalmente, na Seção 5 serão apresentadas as considerações finais dos autores. Entre os requerimentos de *software* para a execução deste minicurso estão o Python, na versão 3, e o NLTK na versão 3.0 ou posterior.

5.2. A abordagem clássica do Processamento de Linguagem Natural

Tradicionalmente, o trabalho no Processamento de Linguagem Natural é decomponível em várias etapas, refletindo as distinções linguísticas teóricas traçadas entre sintaxe, semântica e pragmática.

Essa tentativa de correlação entre uma distinção estratificada (sintaxe, semântica e pragmática) e uma distinção em termos de granularidade (oração versus discurso) às vezes causa alguma confusão ao pensar sobre as questões envolvidas no processamento da linguagem natural. E é amplamente reconhecido que, em termos reais, não é tão fácil separar o processamento da linguagem em caixas correspondentes a cada um dos estratos. No entanto, tal separação serve como uma ajuda pedagógica útil e também constitui a base para modelos arquitetônicos que tornam a tarefa de análise de linguagem natural mais gerenciável do ponto de vista da engenharia de software [Dale, 2010].

Não obstante, a distinção em três partes englobando sintaxe, semântica e pragmatismo apenas serve na melhor das hipóteses como um ponto de partida quando se considera o processamento de textos de linguagem natural real. Uma decomposição desse processo em mais estágios é útil ao se levar em conta o atual estado da arte em combinação com os dados de linguagem reais, como refletidos na Figura 5.1.

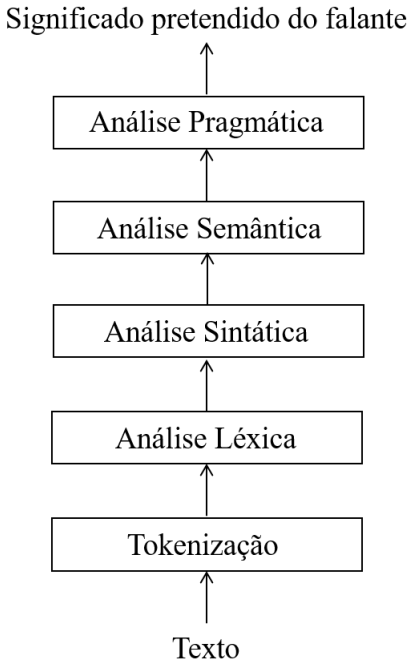

Figura 5.1. Estágios da análise no PLN [Dale, 2010]

Dito isso, as seções a seguir abordam mais detalhadamente cada uma das cinco fases de análise no PLN.

5.2.1. Tokenização

A tokenização, também conhecida como segmentação de palavras, quebra a sequência de caracteres em um texto localizando o limite de cada palavra, ou seja, os pontos onde uma

palavra termina e outra começa [Palmer, 2010]. Para fins de linguística computacional, as palavras assim identificadas são frequentemente chamadas de *tokens*.

Tokenização é bem estabelecida e bem entendida para linguagens artificias tais como linguagens de programação [Aho et al., 2007]. No entanto, tais linguagens artificiais podem ser estritamente definidas para eliminar ambiguidades léxicas e estruturais. Não existe essa 'luxúria' com linguagens naturais, em que os mesmos caracteres podem servir para diferentes fins e em que a sintaxe não é estritamente definida.

Uma diferença fundamental existe entre abordagens de tokenização para **linguagens delimitadas por espaço** e abordagens para **linguagens não segmentadas**. Nas linguagens delimitadas por espaço, tais como a maioria das linguagens europeias, alguns limites de palavra são indicados pela inserção de um espaço em branco. As sequências de caracteres delimitadas não são necessariamente os *tokens* requiridos para processamento adicional, devido tanto à natureza ambígua dos sistemas de escrita como à gama de convenções de tokenização requeridas por diversas aplicações [Palmer, 2010]. Em linguagens não segmentadas, tais como o chinês e o tailandês, as palavras são escritas em sucessão com nenhuma indicação de limite de palavras. A tokenização das linguagens não segmentadas, portanto, requer informação léxica e morfológica adicional.

Em relação à tokenização em linguagens delimitadas por espaço, a maioria das ambiguidades existem entre o uso de sinais de pontuação, tais como ponto final, vírgulas, aspas, apóstrofos e hífens, uma vez que o mesmo sinal de pontuação pode servir para diferentes funções em uma mesma sentença. Considerando o exemplo abaixo, alguns itens de interesse deste estudo são perceptíveis:

> *Compre o apartamento dos seus sonhos na Av. Santo Amaro por apenas R$ 200.000,00 até o próximo dia 24.*

Primeiro, na frase percebe-se o ponto final de três diferentes maneiras: junto aos números para separação de classes numéricas (milhares em R$ 200.000), em abreviações (Av.) e para marcar o final da sentença. Nesta última maneira, já percebe-se uma ambiguidade: o ponto final vem após o número 24, o que poderia confundir com o ponto separador de classes numéricas. O tokenizador deve, assim, estar atento ao uso dos sinais de pontuação e ser capaz de determinar quando um sinal de pontuação é parte de outro token e quando é separado de um *token*. Para resolver esses problemas, deve ser feitas decisões de tokenização sobre a frase em destaque. Por exemplo, deve-se decidir se trataria a frase "*R$ 200.000,00*" diferentemente do que se tivesse sido escrita como "*200 mil reais*" ou "*R$ 200 mil*".

Em vários casos, caracteres tais como vírgulas, ponto e vírgulas e pontos finais devem ser tratados como um *token* separado. Por outro lado, existem outros casos em que esses caracteres devem ser "anexados" a outro *token*. Por exemplo, no português, as abreviações devem conter os pontos, tais como Av. (avenida), Sr. ou Sra. (senhor ou senhora) e Dr. ou Dra. (doutor ou doutora). As abreviações são usadas na linguagem escrita para denotar uma forma mais curta da palavra. Em vários casos, as abreviações são escritas como uma sequência de caracteres terminadas com um ponto final, assim como mostrados no exemplo anterior.

5.2.2. Análise Léxica

A subseção anterior tratava do problema da tokenização, ou seja, quebrar uma entrada de texto em palavras e sentenças que serão tópicos para um processamento subsequente. O processo subsequente em questão é a análise a nível de palavras, a análise léxica. Uma palavra pode ser pensada de duas maneiras: como uma sequência de caracteres no texto em execução, como por exemplo, o verbo ENTREGAR, ou como um objeto mais abstrato que é o termo principal para um conjunto de sequência de caracteres (palavras), sendo o verbo ENTREGAR o objeto abstrato, ou *lemma*, do conjunto {*entrega, entregador, entregando, entregue*}.

A tarefa básica da análise léxica é relacionar variantes morfológicas aos seus *lemmas*, que nada mais são do que as formas canônicas das palavras, ou a forma em que as palavras se encontram no dicionário. Para tal é necessário utilizar um dicionário de *lemmas* em que eles se encontram junto de suas informações semânticas e sintáticas invariantes [Hippisley, 2010]. A lematização é utilizada de diferentes formas, as quais dependem da tarefa a ser realizada pelo sistema de processamento de linguagem natural. A utilização dos *lemmas* é feita, por exemplo, utilizando o verbo ENTREGAR: ENTREGAR + {3ª pessoa do singular, presente}, então, todas as palavras são formadas a partir do *lemma* retirando o sufixo de conjugação.

Segundo [Hippisley, 2010], a análise léxica tem dois lados: o mapeamento da palavra até o seu *lemma*, chamado de *parsing side* e o outro lado é o mapeamento do *lemma* para a palavra, chamado de geração morfológica (do inglês: *morphological generation*). Na recuperação de informação (do inglês: *information retrieval*, IR), o *parsing* e o *generation* servem diferentes propósitos. Para a criação automática da lista de termos chave, faz sentido ter a noção das variantes sobre um único *lemma*. Este propósito é alcançado na prática durante o *stemming*, uma operação de pré processamento de textos onde palavras mais complexas morfologicamente são identificadas, decompondo em seu *stem* invariante, ou melhor, na forma canônica do *lemma* e seus afixos, e no final os afixos são deletados. O *stem*, portanto, é o chamado radical da palavra. Por exemplo, ainda sobre o verbo ENTREGAR, o *lemma* é ENTREGAR e o *stem* é ENTREG, pois a partir do *stem* podem ser criadas outras palavras.

5.2.3. Análise Sintática

Um pressuposto na maioria dos trabalhos na área de PLN é que a unidade básica de análise de significado é a frase: uma frase expressa uma proposição, uma ideia ou um pensamento, além de dizer algo sobre o mundo real ou imaginário. Extrair o significado de uma frase é, portanto, uma questão crucial. As frases não são, no entanto, apenas uma sequência linear de palavras e, por isso, é amplamente reconhecido que para realizar essa tarefa requer uma análise fiel de cada frase, as quais determinam suas estruturas de uma maneira ou de outra. Nas abordagens de PLN baseadas em linguística generativa, isso é geralmente levado a determinar a estrutura sintática ou gramatical de cada frase.

Uma das formas de se representar a análise sintática é por meio de gramáticas e árvores sintáticas (do inglês: *syntax tree* ou *parse tree*). A Figura 5.2 mostra um exemplo de gramática. Em [Moura, 1996] é mostrada a parte formal da construção de uma gramática.

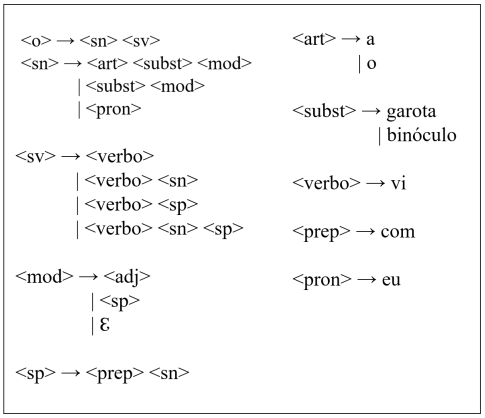

Figura 5.2. Exemplo de uma gramática [Moura, 1996]

A gramática da Figura 5.2 reconhece a frase "*eu vi a garota com o binóculo*" e é composta de dois lados, no qual o lado esquerdo são chamados de símbolos préterminais, onde se decompõem sempre em uma produção, que fica do lado direito, correspondentes ao termos léxicos. A árvore sintática que resultanta do processamento dessa gramática é mostrada na Figura 5.3.

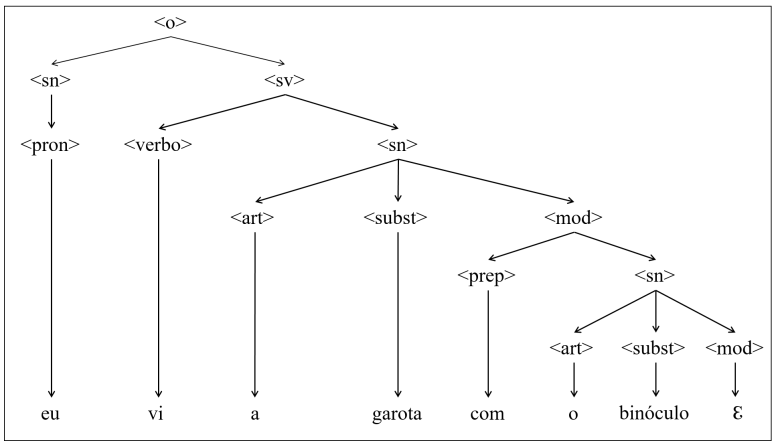

Figura 5.3. Árvore sintática da frase "*eu vi a garota com o binóculo*" [Moura, 1996]

A árvore sintática é a representação de todas as etapas na derivação da frase a partir do nó raiz. Isso significa que cada nó interno na árvore representa a aplicação de uma regra da gramática. Nota-se que a árvore sintática da Figura 5.3 é formada de cima para baixo com a raiz da árvore no topo e as folhas ao fundo.

[o [sn [pron *eu*]] [sv [verbo *vi*] [sn [art *a*] [subst *garota*] [mod [prep *com*] [sn [art *o*] [subst *binóculo*] [mod ε]]]]]]

Figura 5.4. Representação da frase entre colchetes

Outra representação, que é comumente usada na execução de textos, é como uma "*frase entre colchetes*", onde os colchetes são rotulados com as variáveis não-terminais. Um exemplo é mostrado na Figura 5.4.

5.2.4. Análise Semântica

Quando se leva em consideração textos longos, aplicações de PLN específicas podem incluir recuperação de informação, extração de informação, sumarização de textos, mineração de dados e tradução para linguagem de máquina e auxiliares de tradução. A análise semântica também é útil para vários textos menores. Além disso, a análise semântica também tem uma alta relevância nos esforços da comunidade científica para melhorar as ontologias existentes na *Web* e os sistemas de representação de conhecimento [Goddard and Schalley, 2010].

De acordo com [Poesio, 2000], o objetivo final para humanos, bem como para sistemas de PLN, é entender o enunciado – o qual, dependendo das circunstâncias, pode significar a incorporação de informações providas pelo enunciado dentro de sua própria base de conhecimento ou geralmente executando alguma ação em resposta à isso. "Entender" um enunciado é um processo complexo que depende do resultados das análises anteriores (léxica e sintática), assim como das informações léxicas, contexto e do raciocínio comum.

Na linguística, a análise semântica refere-se à análise do significado das palavras, expressões fixadas, sentenças inteiras e enunciados no contexto [Goddard and Schalley, 2010]. Na prática, isso significa traduzir as expressões originais em um tipo de metalinguagem. Em termos gerais, a evidência primária para a linguística semântica vem das interpretações do orador nativo no uso das expressões no contexto (incluindo suas deduções e implicações), padrões de uso, colocação e frequência, que são detectáveis usando técnicas de linguísticas em *Córpus* diversos.

Um requerimento frequentemente identificado para as análises semânticas em PLN é em título de resolução de ambiguidade. Do ponto de vista de uma máquina, vários enunciados humanos estão abertos à multiplas interpretações, pois as palavras podem ter mais de um significado (ambiguidade léxica) ou porque certas palavras, tais como quantificadores, modais ou operadores negativos pode ser aplicadas em diferentes trechos de texto (ambiguidade escopal), ou ainda porque a referência pretendida de pronomes ou outras se referindo à expressões podem não ser fáceis de entender (ambiguidade referencial).

Em relação às ambiguidades léxicas, é normal a distinção entre homônimos (diferentes palavras com a mesma forma, tanto o som quanto a escrita, como por exemplo manga (fruta) e manga (de camisa), ou concerto (sessão musical) e conserto (reparo)) e polissemia (diferentes sentidos para a mesma palavra, por exemplo, "mão suja" e "passaram a mão"). Ambos esses fenômenos são problemáticos em PLN, mas a polissemia apresenta maiores problemas, pois as diferenças de significado em questão, além das diferenças sintáticas associadas e outras diferenças formais são tipicamente mais sutis.

5.2.5. Análise Pragmática

Segundo [Muller, 2003], a análise pragmática foge à estrutura de apenas uma frase, uma vez que ela busca nas outras frases a compreensão do texto que falta àquela frase em análise. Não existem estruturas pré-definidas que atendam a uma representação adequada de problemas de **referências pronomiais**, como por exemplo, os pronomes "la", "seu" e

"o" na frase "João pegou a rosa. Ao pegá-*la*, *seu* espinho *o* espetou", **coerência textual** e **análise do discurso**. Por outro lado, as estruturas mais utilizadas na análise pragmática são as gramáticas baseadas em casos, que são gramáticas semânticas não-terminais para formar padrões, onde caso uma dada frase encaixe na construção padrão, ela poderá ser reconhecida dentro de um contexto [Carbonell and Hayes, 1987, Minker, 1998, Barker, 1998].

Como exemplo de algoritmo utilizado na análise pragmática, [Muller, 2003] discorre sobre um algoritmo proposto por [Lappin and Leass, 1994], de resolução de referências pronomiais. Neste algoritmo, são atribuídos pesos aos pronomes encontrados em frases. Esses pesos são chamados de "valores de saliência" e são atribuídos aos sujeitos encontrados na mesma frase ou nas frases precedentes. Quanto mais próximo do pronome estiver o sujeito, maior será o valor a ele atribuído como referência ao pronome. Após ser atribuído os pesos aos sujeitos de referência encontrados, retiram-se aqueles que não combinam em gênero e número e são levadas em conta outras características como o tipo de pronome, se a frase tem objeto indireto ou advérbio, entre outros fatores.

Há ainda muitos outros algoritmos citados na literatura acerca de resolução de referências pronomiais [Mitkov, 1998, Ng and Cardie, 2002, Haghighi and Klein, 2009], porém, a maioria é dependente de uma estrutura sintática previamente determinada ou pouco flexível a diferentes construções frasais. Por esta dificuldade de ser identificada a construção frasal é que tem sido cada vez mais utilizadas estruturas de preenchimento como *casos* para análise pragmática.

5.3. Python para o Processamento de Linguagem Natural

Textos estão em todos os lugares e, a partir do momento em que aprendemos a ler e escrever, compreendê-los é uma tarefa intuitiva para seres humanos. Porém, dados não estruturados, dos quais os textos são uma grande parcela, representam um dos maiores desafios da indústria de Tecnologia da Informação. Nossa capacidade de absorver e processar esses dados simplesmente não consegue acompanhar a proliferação de documentos na *Web* ou em outros lugares [Feldman and Sanger, 2006]. Como trabalhar com uma grande quantidade de texto em uma linguagem de programação como o Python? Como programadores, é importante entender as estruturas de dados que nos permitem manipular textos como dados não tratados das mais variadas formas.

Trabalhos práticos em Processamento de Linguagem Natural geralmente utilizam grandes corpos de dados linguísticos, chamados de *Córpus* [Bird et al., 2009]. Muitos *Córpus* são projetados para conter uma quantidade cuidadosamente balanceada de material de um ou mais gêneros, o que propicia uma melhor análise estatística e teste de hipótese. Entretanto, antes de trabalharmos com o conceito de *Córpus*, vamos examinar textos a partir de pequenas unidades, como variáveis e listas.

Nesta seção, apresentaremos algumas estruturas de dados básicas de Python para a manipulação de textos, para isso utilizaremos a versão 3 do Python. Você pode baixar e seguir as instruções de instalação do Python 3 em http://www.python.org/downloads/ ou verificar a versão que você já tem instalada executando o interpretador do Python, chamado de *Interactive DeveLopment Environment (IDLE)*. Uma ótima alternativa à instalação do Python e de todos os seus pacotes de forma individual é utilizar a plataforma

Anaconda[1] que inclui o Python 3.6 e todos os pacotes necessários para a execução deste minicurso. Uma outra alternativa é o conjunto de ferramentas *Data Scientist Workbench*[2] da IBM, uma máquina virtual na *Web* com ambientes pré-configurados em linguagens como Python, R e Scala.

Se você optou por utilizar o Python IDLE, você pode encontrá-lo em "*Aplicativos* → *MacPython*" em um *Mac*; em "*Todos os Programas* → *Python*" no *Windows*; ou a partir do *shell* , digitando *idle* (em caso de erro, tente digitar *python*) no Linux. O interpretador imprimirá na tela uma sinopse sobre a versão do *Python*. Basta verificar se você está executando o Python 3.2 ou posterior. Aqui está o resultado do comando "python"no cmd do Windows para a versão 3.4.3:

```
Python 3.4.3rc1 (v3.4.3rc1:69dd528ca625, Feb  8 2015, 11:04:26) on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

O símbolo >>> indica que o IDLE está aguardando alguma entrada. Você pode testá-lo com um comando simples como uma operação matemática.

```
>>> 2 + 5
7
>>>
```

No caso das plataformas *Anaconda* e *Data Scientist Workbench*, você poderá trabalhar diretamente com o *Jupyter Notebook*[3], uma plataforma *Web* que permite a criação e o compartilhamento de código e é integrada ao interpretador do Python (Figura 5.5).

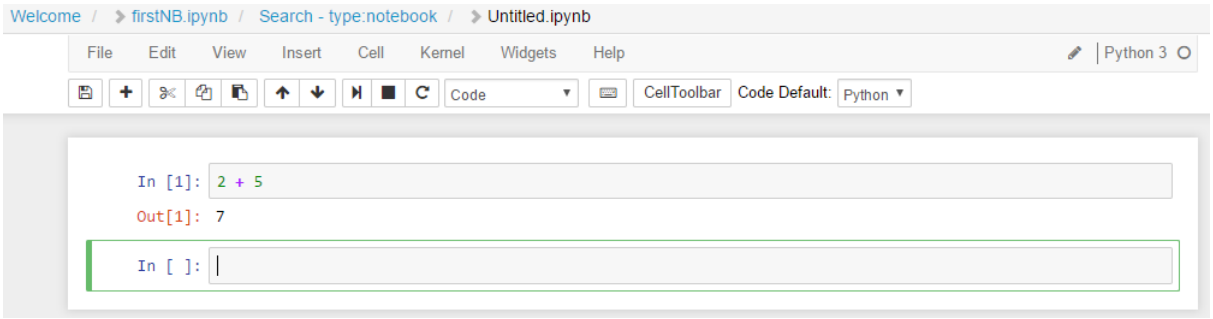

Figura 5.5. Exemplo de código em Python utilizando o Jupyter Notebook

Agora que você já tem o Python configurado, podemos revisar alguns elementos importantes da linguagem para a manipulação de texto.

5.3.1. Conceitos básicos de Python

5.3.1.1. Variáveis

Python é uma linguagem orientada a objetos dinamicamente tipada, ou seja, as variáveis não precisam ser declaradas para serem utilizadas e todas as variáveis em Python são ob-

[1] https://www.continuum.io/downloads
[2] https://my.datascientistworkbench.com/
[3] http://jupyter.org/

jetos. Para utilizar uma variável em Python, é necessário apenas realizar uma **atribuição**, como demonstrada a seguir:

```
>>> temp = 1
>>>
```

Neste exemplo, estamos atribuindo a uma variável chamada de *temp* o valor 1, que é do tipo **inteiro** ou *Int*. Em Python, uma atribuição se dá na forma *"variável = expressão"*, sendo *expressão* um valor de qualquer tipo válido. O nome da variável pode ser qualquer palavra não reservada que inicie com uma letra e pode incluir números e o símbolo de sublinhado (ex.: minha_variavel1).

Podemos atribuir palavras ou textos completos em variáveis em Python. O tipo padrão para dados de texto em Python é chamado de **String**.

```
>>> nome1 = "Carla"
>>> frase1 = 'O rato roeu a roupa do rei de Roma'
>>>
```

5.3.1.2. Strings

Em Python, Strings são identificadas como um conjunto contínuo de caracteres representados entre aspas (que podem ser simples ou duplas). O tipo *Character*, presente em linguagens como *C* e *Java*, não existe em Python. No Python 3 um caractere é na verdade uma String de tamanho 1, representada no sistema de codificação *Unicode*.

```
>>> letraA = "a"
>>> letraB = 'b'
>>>
```

Assumindo que a variável **var1** representa a String 'Amor', a variável **var2** representa a String "Ódio" e a variável **var3** representa a String "Ainda que eu falasse a língua dos anjos", a Tabela 5.1 apresenta algumas possíveis operações de Strings para o Python.

Tabela 5.1. Operações com Strings em Python

Operador	Descrição	Exemplo
+	Concatenação - Cria uma nova String que une o valor de duas o mais Strings.	var1+var2 dará 'AmorÓdio'
*	Repetição - Cria novas Strings, concatenando múltiplas cópias da mesma String.	var1*2 dará 'AmorAmor'
[]	Fatiamento - Retorna o caractere de um dado índice.	var1[1] dará 'm'
[:]	Fatiamento em uma faixa - Retorna os caracteres de uma dada faixa de índices.	var3[1:4] dará 'ind'
in	Adesão - Retorna true se um caracter existe em uma dada String.	'a' in var1 dará true
not in	Adesão - Retorna true se um caracter não existe em uma dada String.	'i' in var1 dará true
len	Tamanho - Retorna o tamanho da String contando espaços e caracteres especiais.	len(var1) dará 4
replace	Substituição - Substitui uma substring de uma String por outra String.	var1.replace('am', 'ia') dará 'iaor'
count	Enumeração - Conta quantas vezes uma substring aparece em uma String.	var1.count('r') dará 1
split	Divisão - Separa uma String em um ponto dado por outra String.	var3.split('língua') daria ['Ainda que eu falasse a ', ' dos anjos']
upper	Maiúsculo - Torna todos os caracteres maiúsculos.	var1.upper() dará 'AMOR'
lower	Minúsculo - Torna todos os caracteres minúsculos.	var1.lower() dará 'amor'

5.3.1.3. Expressões Regulares

Muitas das tarefas de Processamento de Linguagem Natural envolvem correspondência de padrões. Por exemplo, para encontrarmos palavras da mesma família de **biblioteca** em um texto, poderemos encontrar todas as palavras que contém o radical, ou *stem*, **biblio** (como bibliotecário e bibliófilo). Isso também se aplica aos outros morfemas básicos como prefixos e sufixos. Realizar essa tarefa com operações básicas de Strings em Python pode ser uma tarefa complicada, para tal, optamos pelo uso de Expressões Regulares. Para utilizarmos Expressões Regulares em Python, precisamos importar a biblioteca *re*. O trecho de código abaixo usa uma expressão regular (*biblio\w**) para encontrar todas as palavras com o *stem* biblio na string dada:

```
>>> import re
>>> texto = "o bibliotecário abriu a biblioteca"
>>> resultado = re.findall(r'biblio\w*', texto)
>>> resultado
['bibliotecário', 'biblioteca']
>>>
```

Expressões regulares são sequências de caracteres que definem um padrão de busca. Essas sequências são usadas em todas as linguagens de programação, processadores de textos e compiladores. Formalmente, uma expressão regular é uma notação algébrica que caracteriza um conjunto de strings [Jurafsky and Martin, 2000]. Elas são particularmente úteis para a pesquisa em textos, quando há um padrão de pesquisa e um *Córpus* de texto a ser pesquisado. Uma expressão regular irá realizar uma pesquisa em um *Córpus* e retornará todos os textos que correspondem ao padrão. Isso é extremamente útil na identificação de palavras com o mesmo *stem* ou para identificarmos padrões linguísticos em um *Córpus* anotado.

Um exemplo de expressão regular bem simples é aquela que encontra uma palavra dentro de uma string. Por exemplo, podemos verificar a existência da palavra "pessoa" na frase "uma pessoa boa" usando o método *search* do objeto *re* de Python (lembrando que re é a classe que define expressões regulares):

```
>>> import re
>>> texto = "uma pessoa boa"
>>> resultado = re.search(r'pessoa', texto)
>>> resultado
<_sre.SRE_Match object; span=(4, 10), match='pessoa'>
>>>
```

A sintaxe do método *search* é *re.search(**expressão regular**, **string**)*. No exemplo acima, procuramos a palavra "pessoa" em "uma pessoa boa" e o resultado foi um objeto do tipo *_sre.SRE_Match* que especifica a existência da substring "pessoa", além da faixa de caracteres dessa substring na string original e a correspondência. Repare que no campo de expressão regular do método *search* incluímos os caracteres *r'*. Em Python *r'* indica que a string que representa a expressão regular deve ser lida como *raw* (crua), ou seja, símbolos como \, que normalmente indicariam formas especiais de outros caracteres, devem ser lidos como caracteres individuais (dessa forma, \n representa os caracteres

\n e não uma nova linha). Podemos verificar a substring encontrada na busca de duas formas: realizando o fatiamento da string original na faixa especificada ou através do método *_sre.SRE_Match.group(**índice**)* (com índice 0 neste caso em que existe apenas uma correspondência):

```
<_sre.SRE_Match object; span=(4, 10), match='pessoa'>
>>> #continuação do código
>>> texto[4:10]
'pessoa'
>>> resultado.group(0)
'pessoa'
>>>
```

O método *re.search()* retorna apenas a primeira substring que corresponde ao padrão encontrado na string de entrada, para encontrar todas as correspondências presentes na string de entrada, deve-se usar o método *re.findall(**expressão regular**, **string**)*.

Como não é do escopo deste minicurso uma revisão completa de expressões regulares, recomendamos uma leitura complementar para a melhor compreensão desse assunto. A documentação do Python 3[4] apresenta uma ótima revisão sobre expressões regulares, além de uma lista de todos os métodos implementados pela biblioteca *re*.

5.3.1.4. Lista

Além de ser representado como uma String, um texto também pode ser representado como uma sequência de parágrafos, uma sequência de sentenças ou até mesmo como uma sequência de palavras. Para a execução de tarefas de PLN, visualizar um texto como uma sequência de palavras e pontuações é a maneira mais adequada pois podemos trabalhar com uma arquitetura *bottom-up* para identificar o significado do texto a partir do significado de cada palavra, ou seja, analisando os *tokens* do texto. Assim, a String "Casa, comida e roupa lavada." pode ser representada da seguinte forma:

```
>>> texto = ['Casa', ',', 'comida', 'e', 'roupa', 'lavada', '.']
>>>
```

Repare que ao ser representado em uma lista, um texto aparece *tokenizado*, ou seja, conhecemos os limites de cada palavra e símbolo. Como já vimos, esse processo é muito importante para o pré-processamento de PLN. Utilizando expressões regulares, podemos obter uma lista de *tokens* a partir de Strings (utilizando o método re.split()), desde que definamos a condição de separação entre esses *tokens* (ex.: o espaço entre as palavras). Voltando ainda ao que foi dito na seção 2, essa tática nem sempre garante que os *tokens* sejam definidos corretamente, já que existem situações mais complexas, como palavras separadas por hífens, ou números reais divididos por pontos ou vírgulas. Abaixo temos um exemplo de *tokenização* feita com expressões regulares que separa as palavras por espaços, novas linhas e *tabs*. Repare que os símbolos de pontuação aparecem juntos de outras palavras como se fossem um único *token* (ex.: 'memórias.'):

[4]https://docs.python.org/3/library/re.html

```
>>> texto = "Algum tempo hesitei se devia abrir estas memórias."
>>> re.split(r'[ \t\n]+', texto)
['Algum', 'tempo', 'hesitei', 'se', 'devia', 'abrir', 'estas',
'memórias.']
>>>
```

Uma solução para esse problema é a expressão (*r'\w+(?:[-']\w+)*|'|[-.(]+|\S\w*'*). Tente executar o código ***print(re.findall(r'\w+(?:[-']\w+)*|'|[-.(]+|\S\w*', texto))*** e observe a diferença.

Assim como em strings, podemos indexar itens de uma lista utilizando colchetes. Já o método *len(lista)* retorna o número de elementos de uma lista:

```
>>> lista = ['Casa', ',', 'comida', 'e', 'roupa', 'lavada', '.']
>>> lista[2]
comida
>>>len(lista)
7
>>>
```

Por fim, é importante entender como transformar uma lista de strings em uma string única que pode ser exibida na tela ou em um arquivo. Para isso utilizamos o método *.join()* em uma string que servirá de "cola" entre os elementos da lista:

```
>>> bras = ['Viva', 'pois', 'a', 'história']
>>> ' '.join(bras)
'Viva pois a história'
>>> ';'.join(bras)
'Viva;pois;a;história'
>>>
```

5.3.2. O NLTK

Até o presente momento, trabalhamos com ideias para o pré-processamento de textos utilizando apenas conceitos e estruturas básicas de Python. Porém, em problemas práticos é adequado utilizar bibliotecas que ofereçam recursos avançados de PLN. Uma das vantagens do Python é a existência de diversas dessas bibliotecas, como o spaCy[5], o gensim[6], o NLPNET[7] e o NLTK. Para o resto deste minicurso utilizaremos o NLTK, graças à facilidade de uso e à grande quantidade de ferramentas disponíveis, porém, essas bibliotecas não são mutuamente exclusivas e vale a pena conferir o que cada uma delas pode oferecer em termos de uso real de produção.

Criado originalmente em 2001 como parte de um curso de linguística computacional do Departamento de Ciência da Computação e Informação da Universidade da Pensilvânia, o *NLTK* é uma plataforma usada para construir programas *Python* que trabalham com dados de linguagem humana para aplicação em PLN. O NLTK define uma

[5]https://spacy.io/
[6]https://radimrehurek.com/gensim/
[7]http://nilc.icmc.usp.br/nlpnet/

infraestrutura que pode ser usada para construir programas de PLN em *Python*; fornece classes básicas para representar dados relevantes para o processamento da linguagem natural; interfaces padrão para executar tarefas como tokenização, *Part-Of-Speech*, análise sintática e classificação de texto; e implementações padrões para cada tarefa que podem ser combinadas para resolver problemas complexos.

Para a continuação do minicurso é necessária a instalação do NLTK. Se você estiver usando o pacote *Anaconda* ou a máquina virtual do *Data Scientist Workbench*, o NLTK já está instalado. Caso contrário, você precisará realizar o download do NLTK e seguir as instruções presentes em http://www.nltk.org/install.html para o seu sistema operacional. Usuários Windows devem evitar a versão 64-bit do Python, pois essa é notável por causar problemas na instalação do NLTK. Antes de instalar o NLTK, aos usuários da versão IDLE do Python é recomendável instalar a biblioteca NumPy. Para testar se o NLTK está instalado, execute o seguinte código no *prompt* do Python:

```
>>> import nltk
>>>
```

Na ausência de qualquer módulo do NLTK apresentado neste minicurso, você pode fazer o download de todos os módulos do NLTK executando o seguinte código:

```
>>> import nltk
>>> nltk.download()
>>>
```

5.3.2.1. Usando um Corpus e a Classe Text

Até aqui trabalhamos com textos utilizando strings e listas de strings. A partir de agora nossos textos serão extraídos a partir de *Córpus*. O NLTK inclui alguns *Córpus* que possuem utilidades diversas na análise e no processamento de textos. Um exemplo é o *Córpus* **machado**, que inclui uma série de textos do autor Machado de Assis. A sequência de código a seguir demonstra como importar esse *Córpus* para os nossos programas e em seguida lista os *ids* de todos os textos presentes no Córpus.

```
>>> from nltk.corpus import machado
>>> machado.fileids()
['contos/macn001.txt', 'contos/macn002.txt',
'contos/macn003.txt', 'contos/macn004.txt', ...]
```

Cada arquivo corresponde a um dos trabalhos de Machado de Assis. A lista completa pode ser conferida no arquivo README do *Córpus*: *machado.readme()*. Para este minicurso trabalharemos com a obra *Memórias Póstumas de Brás Cubas*.

Assim como em strings do Python, podemos obter um texto de um *Córpus* como uma sequência de caracteres utilizando o método *.raw(id)*. O trecho de código a seguir retorna a obra *Memórias Póstumas de Brás Cubas* e exibe um trecho desse texto em uma faixa que vai do 5600º caractere ao 5800º caractere em utilizando o fatiamento em faixa de strings.

```
>>> raw_text = machado.raw('romance/marm05.txt')
>>> raw_text[5600:5800]
'essa anônima, ainda que não parenta, padeceu mais do
que as parentas.\nÉ verdade, padeceu mais. Não digo
que se carpisse, não digo que se deixasse\nrolar pelo
chão, convulsa. Nem o meu óbito era coisa a'
```

Como já vimos, enxergar um texto como uma sequência de caracteres não é a forma mais eficiente. Por esse motivo, o NLTK também permite retornar um texto como uma lista de *tokens* utilizando o método *.words(id)*:

```
>>> texto1 = machado.words('romance/marm05.txt')
>>> texto1
['Romance', ',', 'Memórias', 'Póstumas', 'de', 'Brás', ...]
>>> len(texto1)
77098
>>>
```

Uma outra possibilidade é utilizar a classe *Text* do NLTK, que além de representar o texto como uma sequência de *tokens*, também implementa uma série de métodos muito úteis:

```
>>> from nltk.text import Text
>>> bras = Text(machado.words('romance/marm05.txt'))
>>>
```

Com a classe *Text* podemos, por exemplo, encontrar uma palavra dentro de diferentes contextos usando o método *.concordance(string)*:

```
>>> bras.concordance('olhos')
'Displaying 25 of 138 matches:
De pé , àcabeceira da cama , com os olhos estúpidos , a b
orelhas . Pela minha parte fechei os olhos e deixei - me i
xões de cérebro enfermo . Como ia de olhos fechados , não
gelos eternos . Com efeito , abri os olhos e vi que o meu
...'
>>>
```

Podemos também encontrar palavras com distribuição de texto similar utilizando o método *.similar(string)*:

```
>>> bras.similar('chegar')
bramanismo acudir leitor soltar com quebrei dali peito aludir
avistar fazer titubear desferirem ver sacrifício casamento já
dizer corpo matrimônio
>>>
```

Podemos encontrar palavras em um contexto com ajuda de expressões regulares:

```
>>> bras.findall("<olhos> (<.*>)")
estúpidos; e; fechados; e; rutilantes; súplices; a; do; ,; babavam;
```

```
na; moles; se; da; umas; .; espraiavam; chamejantes; espetados; ,;
...
>>>
```

Além de *Córpus* de textos como o **machado**, o NLTK também possui *Córpus* anotados como o *MacMorpho* [Fonseca et al., 2015], que podem ser utilizados em tarefas de *part-of-speech tagging*, um tópico que será discutido posteriormente.

5.3.2.2. Stopwords

Stopwords são palavras que podem ser consideradas irrelevantes para o entedimento do sentido de um texto, ou seja, palavras semanticamente irrelavantes. Exemplos: as, e, os, de, para, com, sem, foi. Essas palavras são geralmente removidas de um texto durante a fase de pré-processamento.

O NLTK possui uma lista de *stopwords* para o Português:

```
>>> stopwords = nltk.corpus.stopwords.words('portuguese')
>>> stopwords[:10]
['a', 'ao', 'aos', 'aquela', 'aquelas', 'aquele', 'aqueles',
 'aquilo', 'as', 'até']
 >>>
```

5.3.2.3. Colocações e Bigramas

Em linguística, uma colocação é uma sequência de palavras ou termos que co-ocorrem mais frequentemente do que seria esperado por acaso. Por exemplo, "*vinho tinto*" é uma colocação enquanto "*o vinho*" não é. Uma característica de colocações é que elas são resistentes à substituição com palavras que têm sentidos semelhantes; por exemplo, *vinho marrom* soa definitivamente estranho [Bird et al., 2009]. As colocações são expressões parcial ou totalmente fixas que se estabelecem através do uso repetido dependente do contexto.

Entender e extrair colocações de textos pode ser útil na análise semântica e pragmática. Essa tarefa pode ser facilitada pelo NLTK com o uso de *n-grams*, onde n representa o tamanho da sequência da colocação. Por exemplo, uma sequência de duas palavras é chamada de **bigrama**. O NLTK implementa uma função *bigrams()* que extrai de uma lista de *tokens* uma lista de pares de palavras:

```
>>> from nltk import bigrams
>>> list(bigrams(['meu', 'coração', 'está', 'bem', 'machucado']))
[('meu', 'coração'), ('coração', 'está'), ('está', 'bem'),
('bem', 'machucado')]
>>>
```

Como você pode perceber, colocações são essencialmente bigramas frequentes, exceto que precisamos prestar mais atenção aos casos que envolvem palavras raras. Em

geral, precisamos encontrar bigramas que ocorram mais frequentemente do que seria esperado baseado na frequência de palavras individuais. A função *collocations()* faz exatamente isso:

```
>>> from nltk.corpus import machado
>>> from nltk.text import Text
>>> bras = Text(machado.words('romance/marm05.txt'))
>>> bras.collocations()
Quincas Borba; Lobo Neves; alguma coisa; Brás Cubas; meu pai;
que não; dia seguinte; não sei; Com efeito; que era; Meu pai;
alguns instantes; outra vez; outra coisa; por exemplo; que ele;
mim mesmo; coisa nenhuma; mesma coisa; não era
>>>
```

5.3.2.4. Stemmers

O NLTK também possui o *stemmer* RSLP em Português. O código a seguir demonstra seu funcionamento:

```
>>> stemmer = nltk.stem.RSLPStemmer()
>>> stemmer.stem("copiar")
'copi'
>>> stemmer.stem("paisagem")
'pais'
>>>
```

5.3.2.5. Tokenização

Vimos anteriormente como tokenizar um texto utilizando expressões regulares. O NLTK também possui módulos para tokenização. Podemos *tokenizar* uma sequência de caracteres utilizando o método *.word_tokenize(string)*:

```
>>> import nltk
>>> sentence = """Estou bem, mas não tenho certeza
... se viajarei amanhã às 8:30."""
>>> tokens = nltk.word_tokenize(sentence)
>>> tokens
['Estou', 'bem', ',', 'mas', 'não', 'tenho', 'certeza', 'se',
'viajarei', 'amanhã', 'às', '8:30', '.']
```

5.4. Problemas Práticos

Nesta seção nos apresentamos algumas técnicas importantes da área de PLN que são a base para muitos projetos de pesquisa.

5.4.1. POS-tagging

Um etiquetador é responsável pelo processo de definição da classe gramatical das palavras, de acordo com as funções sintáticas. O processo de classificação das palavras em suas classe gramaticais (a quais chamamos de *Part-of-Speech* ou *POS*) e rotulação são conhecidas como *part-of-speech tagging*, *POS-tagging*, ou simplismente *tagging*.

5.4.1.1. Construindo um tagger

A construção de um *part-of-speech tagger* necessita de um *Corpus* de treinamento anotado. No entanto, os *Corpus* etiquetados podem usar diferentes convenções para a rotulação das palavras, os conjuntos de etiquetas usadas para essa tarefa em particular é conhecida como *tagset*. A Tabela 5.2 mostra o conjunto de etiquetas usadas pelo *MacMorpho*, um *Corpus* em português do Brasil formado por 1,1 milhões de palavras validadas manualmente com anotações morfossintáticas, proposto por [Aluísio et al., 2003].

Tabela 5.2. Etiquetas MacMorpho

CLASSE GRAMATICAL	ETIQUETA
ADJETIVO	ADJ
ADVÉRBIO CONECTIVO SUBORDINATIVO	ADV-KS
ADVÉRBIO RELATIVO SUBORDINATIVO	ADV-KS-REL
ARTIGO (def. ou indef.)	ART
CONJUNÇÃO COORDENATIVA	KC
CONJUNÇÃO SUBORDINATIVA	KS
INTERJEIÇÃO	IN
SUBSTANTIVO	N
SUBSTANTIVO PRÓPRIO	NPROP
NUMERAL	NUM
PARTICÍPIO	PCP
PALAVRA DENOTATIVA	PDEN
PREPOSIÇÃO	PREP
PRONOME ADJETIVO	PROADJ
PRONOME CONECTIVO SUBORDINATIVO	PRO-KS
PRONOME PESSOAL	PROPESS
PRONOME RELATIVO CONECTIVO SUBORDINATIVO	PRO-KS-REL
PRONOME SUBSTANTIVO	PROSUB
VERBO	V
VERBO AUXILIAR	VAUX
SIMBOLO DE MOEDA CORRENTE	CUR

Como as palavras variam de classe gramatical dependendo do contexto em que estão inseridas, vamos trabalhar com dados (etiquetados) em nível de sentença ao invés de palavras. Começaremos por carregar os dados que usaremos:

```
>>> #Importa a biblioteca
>>> import nltk.corpus import mac_morpho
>>> #Carrega as sentença rotuladas do Corpus
>>> sentencas_etiquetadas = mac_morpho.tagged_sents()
```

A etiquetagem mais simples é associar a mesma *tag* para cada *token*. Isso pode parecer um passo insignificante mas estabelece uma *baseline* importante para o desempenho do *tagger*. A fim de conseguir o melhor resultado, etiquetamos cada palavra com a *tag* mais provável/frequente usando o comando:

```
>>> tags = [tag for (word, tag) in mac_morpho.tagged_words()]
>>> nltk.FreqDist(tags).max()
'N'
```

Agora podemos criar um *tagger* que etiqueta qualquer coisa com *N*, considerando que a classe de substantivo é a mais frequente.

```
>>> etiqPadrao = nltk.tag.DefaultTagger('N')
>>> etiqPadrao.tag(tokens)
```

O *tagger* padrão atribui sua *tag* a cada palavra, até mesmo palavras que nunca foram encontradas antes. Como na língua portuguesa a maioria das novas palavras são substantivos, o *tagger* padrão pode ajudar a melhor a robustez do sistema de processamento da linguagem.

5.4.2. N-Gram Tagging

O *unigram tagger* é baseado em um simples algoritmo estatístico no qual para cada *token* é atribuído uma *tag* que seja a mais provável para aquele *token* em particular. Por exemplo, ele vai atribuir a *tag N* a qualquer ocorrência da palavra *Piauí*, desde que *Piauí* seja mais usada como um substantivo do que ser usado como um verbo. Um *unigram tagger* comporta-se apenas como um *tagger* de pesquisa, exceto que há uma técnica mais conveniente para configurá-lo, chamada treinamento. No exemplo de código a seguir, nós treinamos um *unigram tagger*, usando-o para etiquetar uma sentença.

```
>>> import nltk
>>> tagged_sents = nltk.corpus.mac_morpho.tagged_sents()
>>> texto = 'A manhã está ensolarada'
>>> tokens = nltk.word_tokenize(texto)
>>> unigram_tagger = nltk.tag.UnigramTagger(tagged_sents)
>>> unigram_tagger.tag(tokens)
[('A', 'ART'), ('manhã', 'N'), ('está', 'V'), ('ensolarada', 'ADJ')]
```

Por convenção do NLTK, um *token* etiquetado é representado por uma tupla consistindo do *token* e da etiqueta. Aqui nos vemos que *A* é *ART*, um artigo; *manhã* é *N*, um substantivo; *está* é *V*, um verbo (estar); e *ensolarada* é *ADJ*, um adjetivo.

Quando realizamos uma tarefa de processamento de linguagem baseada em unigramas, estamos usando apenas um item do contexto. Nos casos de etiquetagem, consideramos apenas o *token* corrente, isolado de qualquer contexto maior. Dado esse modelo, o melhor que podemos fazer é marcar cada palavra com a sua *tag* a *priori* mais provável. Isso significa que podemos etiquetar um *token* erroneamente, por exemplo, ao atribuir a palavra *cedo* como advérbio independentemente se ela aparece em um contexto como verbo.

Um *n-gram tagger* é uma generalização de um *unigram tagger* cujo contexto é a palavra corrente em conjunto com os n-1 *tokens* precedentes, portanto o *n-gram tagger* seleciona a *tag* que é mais provável no contexto dado. A classe *n-gram tagger* usa um Córpus de treinamento etiquetado para determinar qual *tag* da parte do discurso é mais provável para cada contexto. Aqui vemos casos especiais de *n-gram tagger*, ou seja,

bigram tagger e *trigram tagger*. Primeiramente os treinamos:

```
>>> import nltk
>>> tagged_sents = nltk.corpus.mac_morpho.tagged_sents()
>>> t0 = nltk.DefaultTagger('N')
>>> t1 = nltk.UnigramTagger(tagged_sents, backoff=t0)
>>> t2 = nltk.BigramTagger(tagged_sents, backoff=t1)
>>> t3 = nltk.TrigramTagger(tagged_sents, backoff=t2)
```

Observe que estamos combinando vários *taggers* para aumentar a cobertura e precisão do modelo usando o algoritmo *backoff*. Note também que especificamos o *backoff tagger* quando o etiquetador é inicializado de modo que o treinamento pode tirar proveito dele. Assim, se o *bigram tagger* atribuir a mesma *tag* tal como *unigram tagger* em um determinado contexto, o *bigram tagger* descarta a instância de treinamento, isso mantém o modelo *bigram tagger* o menor possível, a mesma relação também ocorre entre o *trigram tagger* e o *bigram tagger*.

5.4.2.1. Salvando um tagger

O treinamento de um etiquetador em um *Corpus* grande pode tomar um tempo significativo, então é mais conveniente salvar um modelo treinado em um arquivo para mais tarde usa-lo. Vamos salvar nosso *tagger* para o arquivo *mac_morpho.pkl*.

```
>>> from pickle import dump
>>> output = open('mac_morpho.pkl', 'wb')
>>> dump(t3, output, -1)
>>> output.close()
```

5.4.2.2. Carregando um tagger treinado

Agora nos podemos carregar um modelo treinado usando:

```
>>> from pickle import load
>>> input = open('mac_morpho.pkl', 'rb')
>>> tagger = load(input)
>>> input.close()
```

5.4.3. Extração de Informação

Para uma dada pergunta, é provável que alguém já tenha escrito uma resposta para ela em algum lugar. As quantidades de amostras de textos em linguagem natural disponível em formato eletrônico é verdadeiramente surpreendente, e esta aumentando a cada dia. No entanto, a complexidade da linguagem natural pode tornar o acesso às informações nesse texto muito difícil. O estado da arte na PLN ainda está longe de ser capaz de construir representações de propósito geral de significado a partir de texto sem restrições.

As informações vêm em muitos formatos e tamanhos. Um importante formato é o dado estruturado, onde existe uma organização das entidades e relacionamentos regular

e previsível. Por exemplo, talvez podemos estar interessados nas relações entre companhias e localizações. Dada uma companhia em particular, gostaríamos de ser capazes de identificar os locais onde ocorrem os negócios; inversamente, dada uma localização, gostaríamos de descobrir quais empresas fazem negócios nesse local. Se os nossos dados se encontram em formato tabular, então a resposta para essa questão se faz de forma direta. O método para compreender o significado de textos é chamado **extração de informação**, no qual dados não estruturados são convertidos em dados estruturados.

A arquitetura de um sistema simples para a extração de informação consiste em primeiramente dividir o texto bruto de um documento em sentenças usando um segmentador de sentenças, e cada sentença é subdividida em palavras usando um tokenizador. Em seguida, cada sentença é etiquetada com um *POS tagger*, que vai ser muito útil para o próximo passo, **reconhecimento de entidades nomeadas**. Nesse passo, buscamos menções de entidades potencialmente interessantes em cada sentença.

As três primeiras tarefas podem ser executadas facilmente com o conhecimento ja adquerido até aqui usando as instruções:

```
>>> sentences = nltk.sent_tokenize(text) #segmentação em sentenças
>>> sentences = [nltk.word_tokenize(sent) for sent in sentences] #
    segmentação das sentenças em tokens
>>> sentences = [tagger.tag(sent) for sent in sentences] # etiquetagem
    dos tokens
```

5.4.3.1. Chunking

A técnica básica que iremos utilizar para detectar as entidades é chamada de **chunking**, a qual segmenta sequências de multi-tokens. Primeiramente definimos uma gramática chunk, consistindo em regras que indicam como a frase deve ser fragmentada:

```
>>> text = "João comprou um carro esportivo"
>>> tokens = nltk.word_tokenize(text)
>>> tagged = tagger.tag(tokens)
>>> gramatica = r"""PADRAO: {<N><ADJ>+}"""
>>> analiseGramatical = nltk.RegexpParser(gramatica)
>>> analiseGramatical.parse(tagged)
(S João/NPROP comprou/V um/ART (PADRAO carro/N esportivo/ADJ))
```

Para encontrar a estrutura de *chunk* para uma determinada frase, o *RegexpParser* começa com uma estrutura em que nenhum *token* é fragmentado. As regras de fragmentação são aplicadas sucessivamente, atualizando a estrutura do bloco. Depois que todas as regras foram invocadas, a estrutura dos fragmentos resultante é retornada.

Neste exemplo, definimos uma gramática simples com uma única regra usando expressões regulares que procuram *chunks* formados por um substantivo seguindo de um ou mais adjetivos. Usando essa gramática, nós criamos um analisador de *chunk*, e o testamos com nossa sentença, resultando em uma árvore.

5.4.3.2. Reconhecimento de entidade nomeada

Entidades nomeadas (do inglês: named entities, NE) são definidas como frases nomeadas que referem a tipos específicos de indivíduos, como as empresas, pessoas, datas e assim por diante. O objetivo de um sistema de reconhecimento de entidade nomeada (do inglês: Named Entity Recognition, NER) é identificar todas as menções textuais das entidades nomeadas.

No NER, segmentamos e rotulamos as entidades que podem participar de relações interessantes entre si. Tipicamente, serão definidos como frases nominais, ou nomes próprios, conforme exemplificado abaixo:

```
>>> text = "O ministro Edson Fachin do Supremo Tribunal Federal
    determimou a separação do inquérito contra o presidente Michel
    Temer da investigação contra o senador afastado Aécio Neves."
>>> tokens = nltk.word_tokenize(text)
>>> tagged = tagger.tag(tokens)
>>> gramatica = r"""NE: {<NPROP>+}"""
>>> analiseGramatical = nltk.RegexpParser(gramatica)
>>> analiseGramatical.parse(tagged)
(S O/ART ministro/N (NE Edson/NPROP Fachin/NPROP) do/KS (NE
    Supremo/NPROP Tribunal/NPROP Federal/NPROP) determinou/V a/ART
    separação/N do/KS inquérito/N contra/PREP o/ART presidente/N (NE
    Michel/NPROP Temer/NPROP) da/KS investigação/N contra/PREP o/ART
    senador/N afastado/PCP (NE Aécio/NPROP Neves/NPROP) ./.)
```

Em algumas tarefas também é interessante considerar substantivos indefinidos, estes não necessariamente se referem a entidades da mesma forma como definem os nomes próprios, para isso podemos usar os *Chunking* explicados anteriormente.

5.4.3.3. Extração de relações

Finalmente, nas relações de extração, procuramos padrões específicos entre os pares de entidades que ocorrem próximos um do outro no texto, usando esses padrões para construir tuplas registrando as relações entre as entidades.

Uma vez que as entidades nomeadas já tenham sido identificadas no texto, podemos então extrair as relações existentes entre elas. Iremos tipicamente procurar relações entre tipos especificados de entidades nomeadas. Uma maneira de abordar essa tarefa é procurar inicialmente por todas as triplas da forma (X, α, Y), onde X e Y são denominados entidades dos tipos requeridos, e α é a sequência de palavras que intervém entre X e Y. Podemos então usar expressões regulares para extrair apenas aqueles exemplos de α que expressam as relações que estamos procurando.

O exemplo a seguir procura duas NE que estejam relacionadas por PREP, uma preposições. A classe de preposições pertencem as palavras que também funcionam como conectivos, estabelecendo uma relação de dependência (subordinação) entre um sintagma nominal (subordinado) e outro elemento. No exemplo dado podemos observar uma relação de lugar entre *George Silva* e *Romêria*.

```
>>> text = 'A estátua foi criada pelo escultor George Silva, que
    nasceu na Romênia'
>>> tokens = nltk.word_tokenize(text)
>>> tagged = tagger.tag(tokens)
>>> gramatica = r""" NE: {<NPROP>+}
                     REL: {<NE> <.*>* <PREP.*> <.*>* <NE>} """
>>> analiseGramatical = RegexpParser(gramatica)
>>> analise.parse(tagged)
(S A/ART estátua/N foi/VAUX criada/PCP pelo/PDEN escultor/N (REL (NE
    George/NPROP Silva/NPROP) ,/, que/PRO-KS-REL nasceu/V na/PREP|+
    (NE Romênia/NPROP)))
```

5.5. Conclusões

Com tudo o que vimos até aqui, pode-se perceber que as possibilidades proporcionadas pelo Processamento de Linguagem Natural são ilimitadas. Esperamos que com esse minicurso consigamos atrair o interesse de estudantes e profissionais de TI para a área de PLN. Nesta última seção discutiremos algumas aplicações de PLN no mundo real.

As empresas utilizam técnicas de PLN, constantemente para auxiliar na tomada de decisões, principalmente para mensurar a aceitação de determinado produto no mercado. No cenário acadêmico, um dos principais estudos está voltado a descoberta de conhecimento através de técnicas de análise e extração de dados a partir de textos, frases ou apenas palavras. Esse processo envolve a aplicação de algoritmos computacionais que processam textos e identificam informações úteis e implícitas, como os algoritmos Naive Bayes Multinominal para classificação e K-Means para agrupamento; bem como de ferramentas como o NLTK.

Muitos estudos têm surgido direcionados à Extração de Aspectos na Mineração de Opinião utilizando uma abordagem profunda de aprendizado. Nesse tipo de Análise de Sentimentos, a Extração de Aspectos é uma subtarefa a qual tem como função identificar alvos de opinião em textos, ou seja, detectar os aspectos específicos de um produto ou serviço. Essa subtarefa tem fundamental importância nas empresas já que elas usam os resultados obtidos para subsidiar suas estrategias de marketing e apoiar a tomada de decisão sobre determinado produto ou serviço. Outros usuários dessa técnica incluem eventos sociais e movimentos politicos. Diante desses beneficios e de outros ainda em aberto o interesse pela Extração de Aspectos tem aumentado bastante.

Atualmente, as pesquisas de análise de sentimento têm suas aplicações em vários cenários dentre eles estão: *sites* de agregação de opinião como o *booking.com* e *buscapé.com.br*. Não apenas a revisão de produtos e serviços como também questões políticas e sociais usam Extração de Aspectos. A inteligencia dos negócios é uma das principais beneficiárias da Extração de Aspectos na Mineração de Opinião.

Outro problema complexo que merece atenção especial é a Sumarização. A Sumarização Automática de Textos é uma das etapas de Mineração de Textos que permite que, a partir da versão resumida de vários textos ou da opinião de vários autores, consigamos sumarizar uma ideia principal. O uso de PLN revela-se benéfico ao processo de sumarização, principalmente quando se processam documentos sem nenhuma estrutura

e/ou padrão definido. Dentre as variações do processo de sumarização, as técnicas extrativas são as mais bem estudadas até o momento, em sua maioria suportando o idioma inglês, com poucas variações de suporte a mais um idioma. Outro problema que aguarda solução é o tratamento de determinada doença a partir de dados não estruturados obtidos através de relatórios médicos, como exemplo podemos citar o tratamento de determinado câncer a partir da análise de um Córpus médico.

Diante da relevância das técnicas de Processamento de Linguagem Natural, hoje, não só no ambiente acadêmico mais também na área de negócios, faz-se necessário que outros integrantes venham contribuir com novas ideias nessa área bastante promissora.

Referências

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools, Second Edition*. Addison-Wesley Longman Publishing Company.

[Aluísio et al., 2003] Aluísio, S., Pelizzoni, J., Marchi, A. R., de Oliveira, L., Manenti, R., and Marquiafável, V. (2003). *An Account of the Challenge of Tagging a Reference Corpus for Brazilian Portuguese*, pages 110–117. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Barker, 1998] Barker, K. (1998). *Semiautomatic Recognition of Semantic Relationships in English Technical Texts*. PhD thesis.

[Bird et al., 2009] Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition.

[Carbonell and Hayes, 1987] Carbonell, J. G. and Hayes, P. J. (1987). *Robust Parsing Using Multiple Construction-Specific Strategies*, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Dale, 2010] Dale, R. (2010). Classical approaches to natural language processing. In *Handbook of natural language processing*. Chapman and Hall/CRC, 2nd edition.

[Feldman and Sanger, 2006] Feldman, R. and Sanger, J. (2006). *Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, New York, NY, USA.

[Fonseca et al., 2015] Fonseca, E. R., G Rosa, J. L., and Aluísio, S. M. (2015). Evaluating word embeddings and a revised corpus for part-of-speech tagging in portuguese. *Journal of the Brazilian Computer Society*, 21(1):2.

[Goddard and Schalley, 2010] Goddard, C. and Schalley, A. C. (2010). Semantic analysis. In *Handbook of natural language processing*. Chapman and Hall/CRC, 2nd edition.

[Haghighi and Klein, 2009] Haghighi, A. and Klein, D. (2009). Simple coreference resolution with rich syntactic and semantic features. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 3*, pages 1152–1161.

[Hippisley, 2010] Hippisley, A. (2010). Lexical analysis. In *Handbook of natural language processing*. Chapman and Hall/CRC, 2nd edition.

[Indurkhya and Damerau, 2010] Indurkhya, N. and Damerau, F. J. (2010). *Handbook of Natural Language Processing*. Chapman & Hall/CRC, 2nd edition.

[Jurafsky and Martin, 2000] Jurafsky, D. and Martin, J. H. (2000). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

[Lappin and Leass, 1994] Lappin, S. and Leass, H. J. (1994). An algorithm for pronominal anaphora resolution. *Computational Linguistics*, 20(4):535–561.

[Liu and Zhang, 2013] Liu, B. and Zhang, L. (2013). *A survey of opinion mining and sentiment analysis*, pages 415–463. Springer US.

[Minker, 1998] Minker, W. (1998). Stochastic versus rule-based speech understanding for information retrieval. *Speech Communication*, 25(4):223 – 247.

[Mitkov, 1998] Mitkov, R. (1998). Robust pronoun resolution with limited knowledge. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics*, pages 869–875.

[Moura, 1996] Moura, R. S. (1996). Sos: Um sistema orto-sintático para a língua portuguesa. Master's thesis, Universidade Federal de Pernambuco.

[Muller, 2003] Muller, D. N. (2003). Processamento de linguagem natural. Technical report, Universidade Federal do Rio Grande do Sul.

[Ng and Cardie, 2002] Ng, V. and Cardie, C. (2002). Improving machine learning approaches to coreference resolution. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 104–111.

[Palmer, 2010] Palmer, D. D. (2010). Text preprocessing. In *Handbook of natural language processing*. Chapman and Hall/CRC, 2nd edition.

[Poesio, 2000] Poesio, M. (2000). Semantic analysis. In *Handbook of natural language processing*. Marcel Dekker.

[Robertson, 1946] Robertson, D. W. (1946). A note on the classical origin of "circumstances"in the medieval confessional. *Studies in Philology*, 43(1):6–14.