

FIAP

NBA



Introdução e Técnicas de Pré-processamento

Dheny R. Fernandes

1. Introdução

1. Definição do problema
2. Exemplos

2. Técnicas de Pré-processamento

- Tokenização e N-grama
- Regex
- Stop-words
- Normalização de Texto
- Stemmer
- Lemmatizer
- POS-Tagger
- Além de NLTK

3. Exercícios

Introdução



Num sentido amplo, Processamento de Linguagem Natural (NLP) trata de qualquer tipo de **manipulação computacional de linguagem natural**, desde uma simples **contagem de frequências de palavras** para comparar diferentes estilos de escrita, até o “**entendimento**” **completo de interações humanas** (pelo menos no sentido de oferecer uma resposta útil à eles).

As tecnologias baseadas em NLP estão se tornando cada vez mais **pervasivas** e, diante das interfaces homem-máquina mais naturais e meios mais sofisticados de armazenamento de informações, processamento de linguagem tem alcançado um papel central numa sociedade da informação multi-lingual.

Muitos são os exemplos de aplicações de NLP. Aqui estão alguns:

- Reconhecimento de escrita à mão ([paper](#))
- Search engines (google, bing)
- Machine translation (google translate)
- Chatbots (many chat)

Técnicas de Pré-processamento

Como sabemos, os computadores lidam com números. Qualquer outro tipo de dado (como áudio, texto, imagens, etc.) precisa passar por um processo de transformação para números.

Para realizar essa transformação, antes precisamos fazer o pré-processamento de dados, cujo objetivo é preparar os textos para criar um espaço de características adequado.

O primeiro passo é a tokenização dos dados

A tokenização nada mais é que uma **sequência de “n” elementos de uma sequência maior, denominadas n-gramas**. Os tipos de n-grama são definidos pela quantidade de elementos que os compõem. Unigramas ($N = 1$), Bigramas ($N = 2$) e Trigramas ($N = 3$).

A ideia consiste em dividir o texto, geralmente usando “espaço” como separador, em tokens no intuito de realizar outras atividades

Vejamos alguns exemplos:

Considere as frases:

Eu gosto de assistir jogos de futebol

Já eu, prefiro assistir jogos de basquete

Unigrama:		0	1
	assistir	1	1
	basquete	0	1
	de	2	1
	eu	1	1
	futebol	1	0
	gosto	1	0
	jogos	1	1
	já	0	1
	prefiro	0	1

Bigrama

	0	1
assistir jogos	1	1
de assistir	1	0
de basquete	0	1
de futebol	1	0
eu gosto	1	0
eu prefiro	0	1
gosto de	1	0
jogos de	1	1
já eu	0	1
prefiro assistir	0	1

Trigrama

	0	1
assistir jogos de	1	1
de assistir jogos	1	0
eu gosto de	1	0
eu prefiro assistir	0	1
gosto de assistir	1	0
jogos de basquete	0	1
jogos de futebol	1	0
já eu prefiro	0	1
prefiro assistir jogos	0	1

Para realizar a tokenização, podemos implementar uma solução própria ou usar a biblioteca NLTK. NLTK, ou Natural Language Toolkit, é uma plataforma para construir programas em Python para trabalhar com dados de linguagem humana.

Vamos usá-la para realizar a maioria das tarefas de pré-processamento que compõe o pipeline de transformação de dados textuais.

No código, deixei um exemplo de como funciona a tokenização usando NLTK.

Expressão regular é uma maneira de **identificar padrões em sequências de caracteres**. No Python, o módulo **re** provê um analisador sintático que permite o uso de tais expressões. Os padrões definidos através de caracteres que tem significado especial para o analisador.

Principais caracteres:

- Ponto (.): Em modo padrão, significa qualquer caractere, menos o de nova linha.
- Circunflexo (^): Em modo padrão, significa inicio da *string*.
- Cifrão (\$): Em modo padrão, significa fim da *string*.
- Contra-barras (\): Caractere de escape, permite usar caracteres especiais como se fossem comuns.
- Colchetes ([]): Qualquer caractere dos listados entre os colchetes.
- Asterisco (*): Zero ou mais ocorrências da expressão anterior.
- Mais (+): Uma ou mais ocorrências da expressão anterior.
- Interrogação (?): Zero ou uma ocorrência da expressão anterior.
- Chaves ({n}): n ocorrências da expressão anterior.
- Barra vertical (|): “ou” lógico.
- Parenteses (()): Delimitam um grupo de expressões.
- \d: Dígito. Equivale a [0-9].
- \D: Não dígito. Equivale a [^0-9].
- \s: Qualquer caractere de espaçamento ([\t\n\r\f\v]).
- \S: Qualquer caractere que não seja de espaçamento.([^\t\n\r\f\v]).
- \w: Caractere alfanumérico ou sublinhado ([a-zA-Z0-9_]).
- \W: Caractere que não seja alfanumérico ou sublinhado ([^a-zA-Z0-9_]).

```
import re
rex = re.compile('\w+') #qualquer caracter alfanumérico - compilado
bandas = 'Queen, Aerosmith & Beatles'
print (bandas, '->', rex.findall(bandas))
phone = "2004-959-559 # This is Phone Number"
num = re.sub('#.*$', "", phone) #elimina tudo após #
print ("Phone Num : ", num)
num = re.sub(r'\D', "", phone) # só deixa número
print ("Phone Num : ", num)
```

Fácil?

O que esse regex faz?

```
(?<=@)[^.]+(?=\.)
```


Regex nem sempre é a melhor opção, no entanto. No código eu comparo duas abordagens.

Na prática, procure sempre a solução mais pythônica.

Refere-se ao conjunto de palavras irrelevantes ou que não contribuem para o significado da frase. Normalmente é composto por artigos, advérbios, preposições e alguns verbos.

NLTK apresenta um conjunto de palavras que podemos usar para tratar textos removendo stop-words. No código eu apresento eles.

Quando lidamos com texto, o espaço de características será composto por palavras (isso numa simples representação vetorial de texto). Assim, é importante tratar o texto de forma que as palavras “Que” e “que” não sejam tratadas de maneira diferentes.

Para isso, usamos normalização, que consiste em aplicar várias técnicas no texto, a saber: lowercasing, stemming, lemmatization

Para entender o funcionamento dessas três técnicas, considere os exemplos abaixo:

- O carro que estava quebrado voltou a funcionar
- Meu carro quebrou e não está funcionando

Realizando a vetorização dessas frases, eu obterei a seguinte representação:

	0	1
carro	1	1
estava	1	0
está	0	1
funcionando	0	1
funcionar	1	0
meu	0	1
não	0	1
que	1	0
quebrado	1	0
quebrou	0	1
voltou	1	0

Removendo as stop-words, teremos a seguinte representação:

	0	1
carro	1	1
funcionando	0	1
funcionar	1	0
quebrado	1	0
quebrou	0	1
voltou	1	0

Entretanto, ainda temos variações de uma mesma palavra dentro do meu conjunto de características:

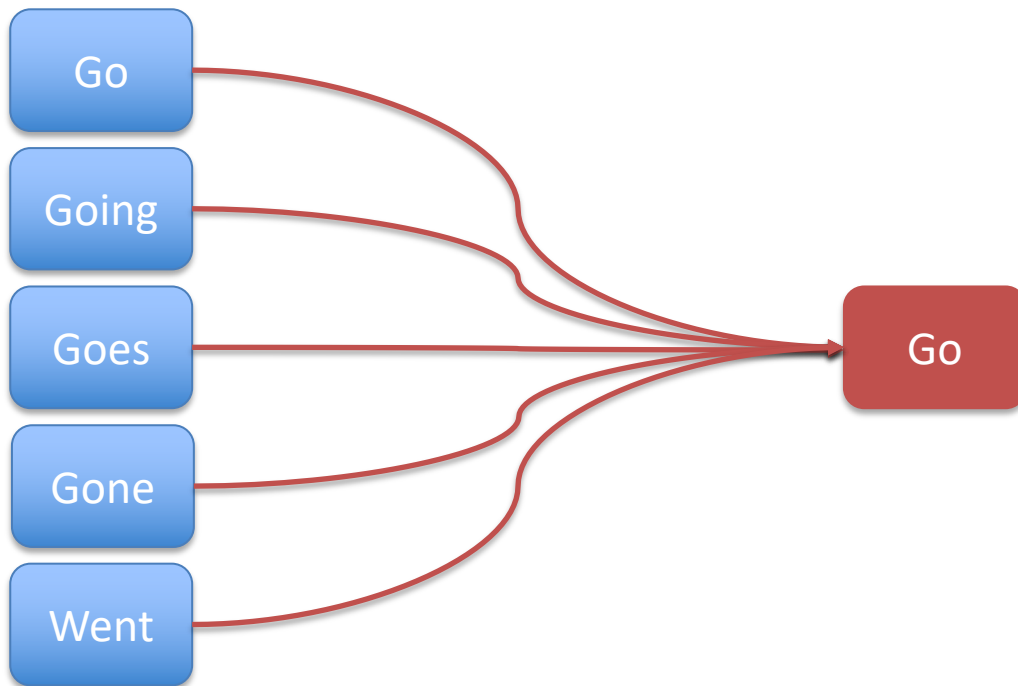
- Funcionar: funcionar, funcionando
 - Quebrar: quebrado, quebrou
-
- Podemos melhorar a representação de texto usando stemming e lemmatization

Ambas possuem como objetivo transformar palavras/tokens num formato padrão.

O que diferencia cada uma delas é a maneira com que elas atingem esse objetivo.

Vamos entender:

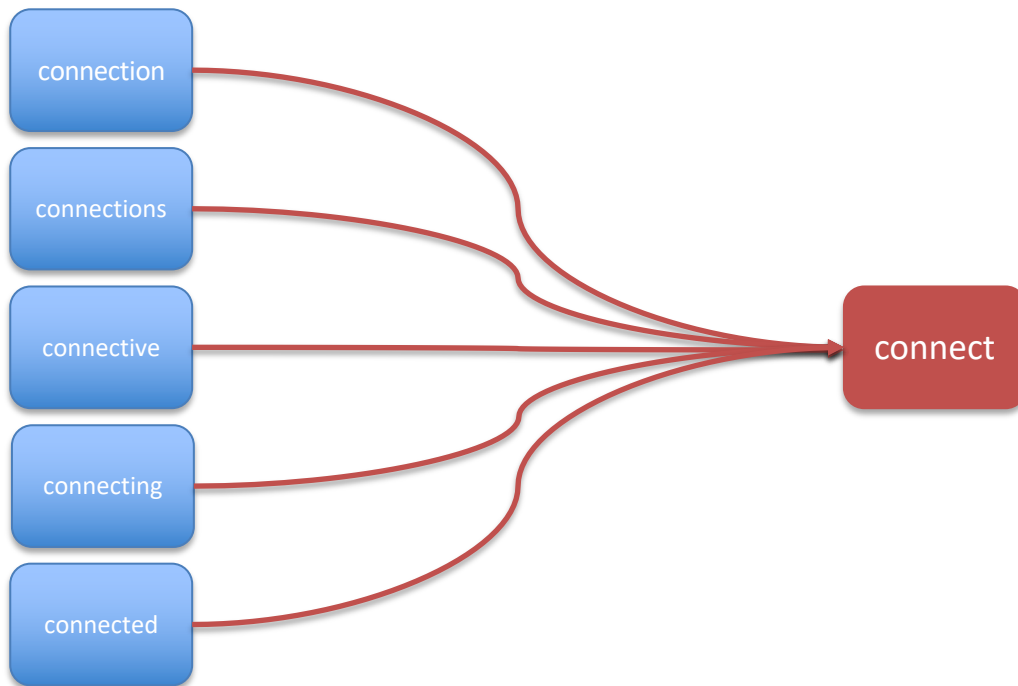
O lemmatization é o processo de determinar a raiz de uma palavra, não importando sua variação. Veja o exemplo:



Entretanto, o NLTK não oferece suporte para lemmatizing em português. Assim, vamos usar a biblioteca Spacy. Voltaremos nela mais à frente.

No código, eu mostro como instalar a biblioteca e usar o lemmatizing.

O segundo método que temos é o stemming, que é uma versão mais simples do lemmatizing que consiste na remoção do sufixo de uma palavra. Observe o exemplo:



Entretanto, de maneira direta, o stemming não funciona de maneira adequada para o português. Para resolver esse caso, podemos usar uma biblioteca diferente do NLTK. No código, eu mostro como fazer.

Vimos várias técnicas de normalização, mas qual o objetivo de usá-la? Comentamos sobre o espaço de características no início da discussão. Esse é o resultado final

	0	1
carro	1	1
estava	1	0
está	0	1
funcionando	0	1
funcionar	1	0
meu	0	1
não	0	1
que	1	0
quebrado	1	0
quebrou	0	1
voltou	1	0



	0	1
carr	1	1
est	1	1
funcion	1	1
quebr	1	1
volt	1	0

Na escola primária, aprendemos a diferença entre substantivo, verbos, advérbios e adjetivos. Tais classes gramaticais são categorias úteis para muitas tarefas de processamento de linguagem natural.

Aqui, teremos os seguintes objetivos:

- Quais são as categorias léxicas (classes gramaticais) e como elas são usadas em NLP
- Uma boa estrutura de dados em Python para armazenar palavras e suas categorias
- Como podemos marcar (taguear) automaticamente cada palavra de um texto com sua classe

O processo de classificar palavras em classes gramaticais é chamado de *Part-of-speech tagging*, ou POS-Tagging.

Geralmente, é o segundo passo num pipeline de NLP, seguido após o tratamento de texto que vimos anteriormente.

Aqui, vamos usar tanto NLTK quanto Spacy, seja em inglês ou português, além de treinar um POS-tagger visando marcação automática.

O exemplo a seguir ilustra a ideia:

```
text1 = nltk.word_tokenize("They refuse to permit us to obtain the refuse permit")  
nltk.pos_tag(text1)
```

```
[('They', 'PRP'),  
 ('refuse', 'VBP'),  
 ('to', 'TO'),  
 ('permit', 'VB'),  
 ('us', 'PRP'),  
 ('to', 'TO'),  
 ('obtain', 'VB'),  
 ('the', 'DT'),  
 ('refuse', 'NN'),  
 ('permit', 'NN')]
```

Verbo
(recusar)

Verbo
(permitir)

Substantivo
(Lixo/entulho)

Substantivo
(licença)

Mas, afinal, qual a real utilidade disso?

Considere as seguintes palavras:

- Woman (substantivo)
- Bought (verbo)
- Over (preposição)
- The (determinante)

Com o POS-tagging, é possível encontrar palavras que pertençam à mesma classe gramatical.

```
nltk.download('brown')
text = nltk.Text(word.lower() for word in nltk.corpus.brown.words())
text.similar('woman')
```

```
[nltk_data] Downloading package brown to
[nltk_data] C:\Users\Dheny\AppData\Roaming\nltk_data...
[nltk_data] Unzipping corpora\brown.zip.
man time day year car moment world house family child country boy
state job place way war girl work word
```

```
text.similar('bought')
```

```
made said done put had seen found given left heard was been brought
set got that took in told felt
```

```
text.similar('over')
```

```
in on to of and for with from at by that into as up out down through
is all about
```

```
text.similar('the')
```

```
a his this their its her an that our any all one these my in your no
some other and
```

Com isso, conseguimos treinar um tagger para taggear palavras novas.

Entretanto, NLTK não possui suporte nativo ao português, mas é possível fazer o download de um Corpus para resolver nosso problema.

Aqui, vamos usar o Corpus Floresta:

- O projeto Floresta Sintá(c)tica é uma colaboração entre a Linguateca e o projecto VISL. Contém textos em português (do Brasil e de Portugal) anotados (analísados) automaticamente pelo analisador sintático PALAVRAS e revistos por linguistas.

Veremos que a tag que for atribuída a uma palavra irá depender da própria palavra e seu contexto numa sentença. Assim, o tagueamento ocorre a nível de sentença, e não de palavra.

Vamos analisar as seguintes estratégias de taguemanto:

- Default Tagger
- Unigram Tagger
- Bigram Tagger
- Uma combinação entre eles

O Default Tagger atribui a mesma tag para cada token. Apesar de parecer simplória, essa técnica estabelece um importante **baseline** para o desempenho do tagueador.

Para isso, eu preciso descobrir a tag mais **frequente** num Corpus e, de posse dessa informação, crio um tagueador que atribuirá a todos os tokens essa tag.

No código eu mostro como fazer isso e, além disso, criamos um conjunto de treino e teste para podermos avaliar as diferentes estratégias de tagging.

Como esperado, o desempenho do Default Tagger foi muito aquém do esperado, já que atribui a mesma tag para todas as palavras.

Entretanto, estatisticamente, e muito por conta do Corpus que estamos usando, quando tagueamos milhares de palavras num texto, a maioria das novas serão de fato substantivos.

Dessa maneira, **utilizar o Default Tagger pode ajudar a melhorar a robustez de um sistema de processamento de linguagem.**

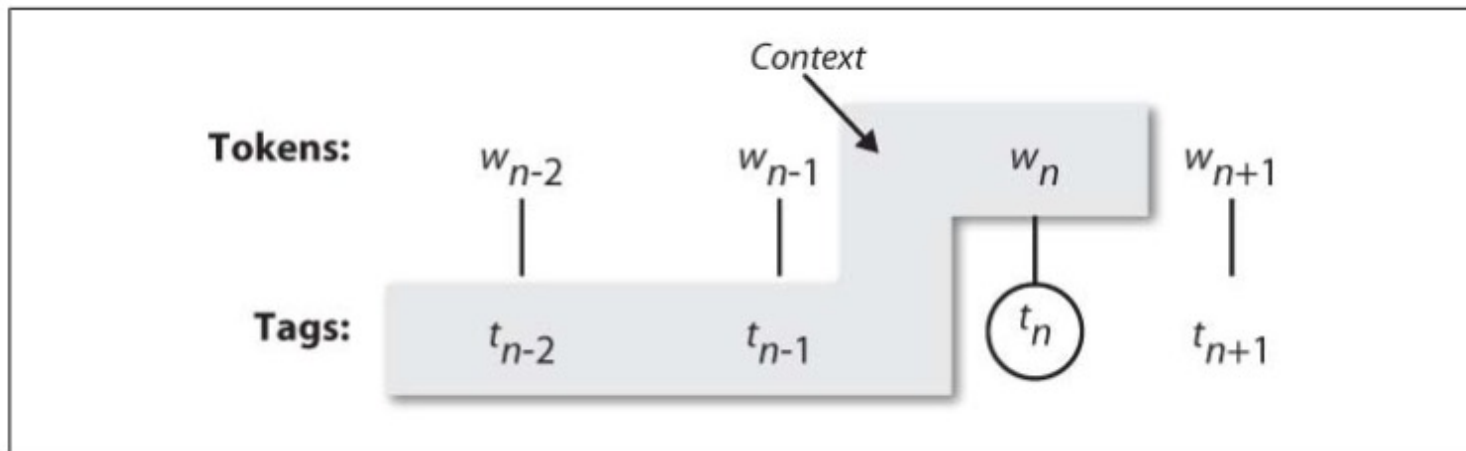
A segunda abordagem que temos é o Unigram Tagger, que se baseia em frequência estatística da classe gramatical mais vezes atribuída a uma palavra.

Em outras palavras, o Unigram Tagger estabelece a tag mais provável por olhar para uma palavra, encontrar suas diferentes funções sintáticas dentro do Corpus, pegar aquela cuja recorrência seja máxima e atribuir essa tag para ocorrências dessa palavra no conjunto de teste.

No código, é possível ver que a performance melhorou substancialmente usando essa abordagem.

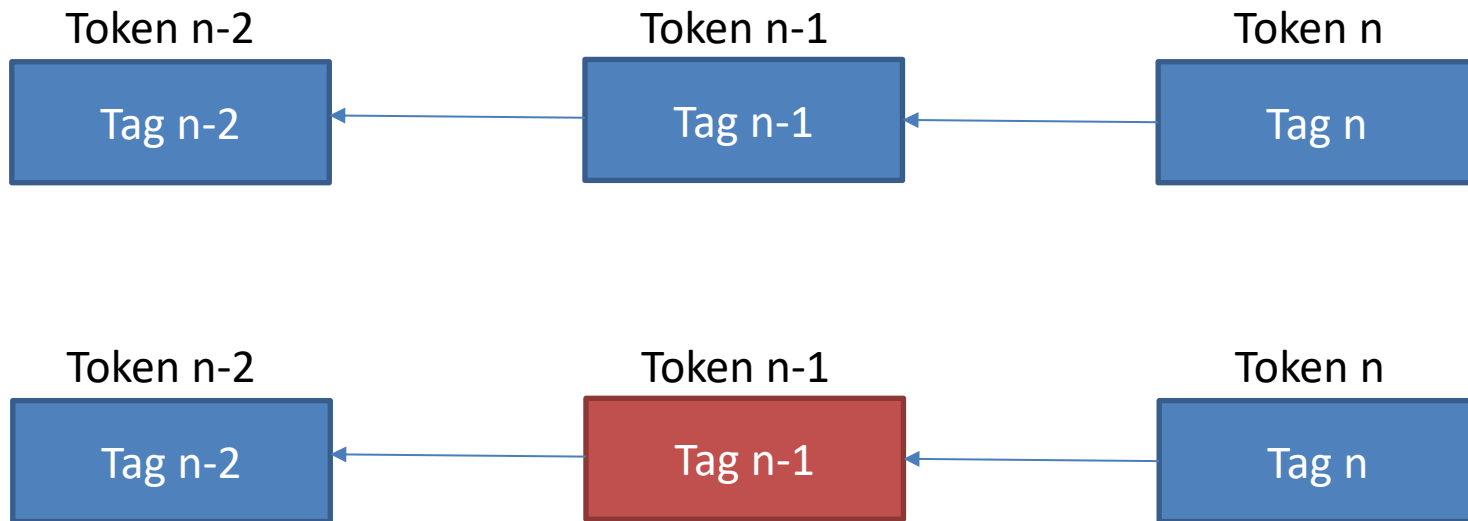
O conceito de Unigram Tagger pode ser generalizado para N-gram Tagger. Aqui vamos ver Bigram Tagger.

O N-gram Tagger possui um contexto que é definido pelo token atual em conjunto com as tags dos $n-1$ tokens antecedentes. Observe a imagem abaixo:



No código, eu treinei um modelo usando um Bigram Tagger. Vamos analisar seus resultados.

Por qual motivo os resultados foram ruins?



À medida que n aumenta, a especificidade dos contextos aumenta, assim como a chance de que os dados que desejamos marcar contêm contextos que não estavam presentes nos dados de treinamento.

Isto é conhecido como **problema esparsidade dos dados** e é bastante recorrente em NLP.

Como consequência, existe um *trade-off* entre acurácia e a cobertura dos resultados (e isto é relacionado com o *trade-off* de *precision/recall* em recuperação de informação)

Uma maneira de resolver o *trade-off* entre acurácia e cobertura é utilizar o algoritmo com melhor acurácia que temos, mas retornar a algoritmos com maior cobertura quando necessário.

Por exemplo, podemos combinar o resultado de um Bigram Tagger, Unigram Tagger e Default Tagger da seguinte maneira:

- Tente taggear o token com o Bigram Tagger
- Se ele falhar, tente usar Unigram Tagger
- Se ele também falhar, use o Default Tagger

Para isso, usamos o conceito de *backoff* quando declaramos um Tagger.

Treinar um tagger pode consumir tempo considerável num Corpus muito grande.

Ao invés de treinar um tagger toda vez que precisarmos de um, é conveniente salvar um tagger treinado para posterior reuso. Depois, é possível carregar o modelo treinado e usá-lo em novos dados.

No código, eu mostro isso:

Apesar de ser a biblioteca mais importante, a NLTK não é a única opção para trabalhar em Python. Vamos conhecer, minimamente, outras duas opções:

- TextBlob
- Spacy

TextBlob foi criado com base nas libs NLTK e Pattern e tem por objetivo prover uma interface simples para as funções do NLTK.

No código eu deixei um exemplo de como usar TextBlog para classificar sentimentos.

Abaixo, deixo o link da documentação oficial bem como um artigo que a autora ensina a fazer processamento de texto usando TextBlob:

- [Documentação](#)
- [Artigo](#)

TextBlob foi criado com base nas libs NLTK e Pattern e tem por objetivo prover uma interface simples para as funções do NLTK.

No código eu deixei um exemplo de como usar TextBlog para classificar sentimentos.

Abaixo, deixo o link da documentação oficial bem como um artigo que a autora ensina a fazer processamento de texto usando TextBlob:

- [Documentação](#)
- [Artigo](#)

SpaCy é outra opção recente no campo de NLP mas vem ganhando espaço na indústria. Com uma abordagem mínima e otimizada, seu foco é a simplicidade e desempenho. Ao contrário do NLTK, o SpaCy traz em sua API apenas uma opção de algoritmo (em teoria, o melhor) para cada finalidade. Ele é construído com Cython e, por isso, é muito rápido.

Uma das grandes vantagens do SpaCy é que ele tem modelos treinados em português disponíveis para download. Com o modelo da língua portuguesa carregado acessar o POS Tag dos tokens é tão simples quanto acessar um atributo.


No código, eu apresento um exemplo usando o SpaCy.

Exercícios



Obrigado!

profdheny.fernandes@fiap.com.br

 /dhenyfernandes

FIAP MBA⁺

Copyright © 2022 | Professor Dheny R. Fernandes

Todos os direitos reservados. Reprodução ou divulgação total ou parcial deste documento, é expressamente proibido sem consentimento formal, por escrito, do professor/autor.

FIAP