

Relatório 24 - Processamento de Linguagem Natural (NLP)

Guilherme Loan Schneider

Descrição da atividade

O primeiro passo para fazer o computador compreender relações entre palavras consiste na Tokenização. Esse processo consiste em transformar palavras individuais em Tokens únicos, ou seja, imagine que uma frase foi tokenizada e nessa frase havia a palavra “andar”, ela possui um ID único e toda vez que essa palavra aparecer em outra frase, ela possuirá esse mesmo token.

À medida que mais frases são inseridas no dataset, o algoritmo consegue compreender o sentido que essas palavras estão fazendo ali, criando uma relação entre elas. Esse tipo de aplicação é comumente feito em classificação de e-mails, como spam ou não spam.

Na imagem abaixo, podemos verificar cinco tokens na frase “what is machine learning?” representados por cores diferentes. Nesse caso, a palavra “what” pode ser representada pelo token 001, “is” por 002, “machine” por 003, “learning” por 004 e “?” por 005.

Tokens

5

Characters

25

Diagrama de tokenização da frase "what is machine learning?". A frase é exibida com cada palavra e o símbolo de interrogação em um bloco de cor diferente: "what" (roxo), "is" (verde), "machine" (laranja) e "learning?" (rosa). O bloco "learning?" inclui o ponto de interrogação. O diagrama ilustra como a frase é dividida em tokens individuais para o processamento de linguagem natural.

As frases do dataset serão formadas por conjuntos de tokens, resultando em uma sequência, similar a um vetor: [001, 002, 003, 004, 005]. Essa sequência representa a frase citada anteriormente.

Existem ainda alguns problemas que podemos enfrentar quando lidamos com esse tipo de dado, como tamanhos de frases diferentes a serem passado a uma rede neural, o que acontece quando inserimos frases com palavras nunca antes vistas, dentre outros.

O Tensorflow possui bibliotecas capazes de solucionar esses problemas, podemos utilizar o “padding” para uniformizar o tamanho das frases. Esse parâmetro adiciona zeros a esquerda ou direita, da sequência de tokens (frase), pegando a maior sequência de tokens como valor de referência para padronizar.

```
[[ 0  0  0  5  3  2  4]
 [ 0  0  0  5  3  2  7]
 [ 0  0  0  6  3  2  4]
 [ 8  6  9  2  4 10 11]]
```

Existe também um parâmetro chamado de `oov_token`, que adiciona um valor definido (geralmente uma palavra extremamente difícil de ser vista) como valor default caso apareça uma palavra nunca antes vista.

```
[[ 5, 1, 3, 2, 4], [2, 4, 1, 2, 1]]

{'think': 9, 'amazing': 11, 'dog': 4, 'do': 8, 'i': 5, 'cat': 7,
 'you': 6, 'love': 3, '<OOV>': 1, 'my': 2, 'is': 10}
```

Utilizando um dataset para prever frases sarcásticas ou não, realizamos os mesmos processos citados anteriormente, chegando em uma sequência de tokens no seguinte formato:

```
[16004  355  3167  7474  2644      3  661  1119      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0]
(28619, 152)
```

Esse, por sua vez, tem um tamanho muito maior, dado que é levado em conta o tamanho da maior frase no dataset, que conterá 152 palavras.

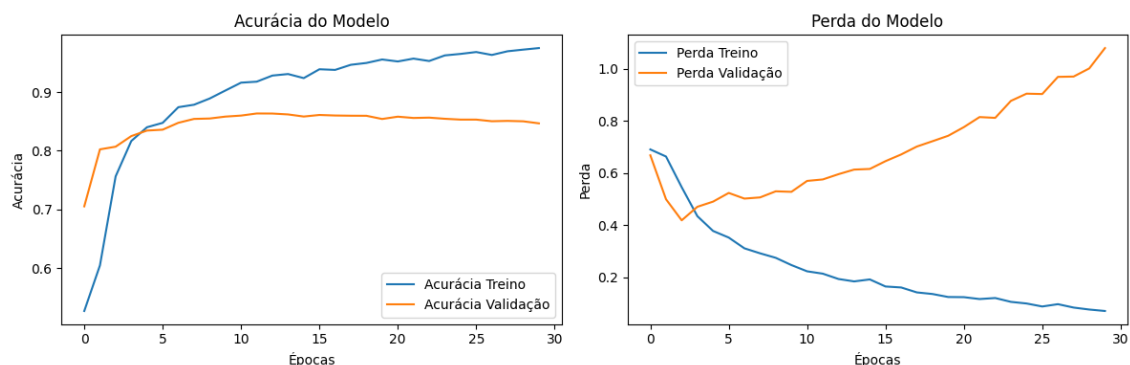
Em seguida, realizamos a divisão do dataset em teste e treino, tanto as sequências quando as labels (se é sarcástica ou não). Também criamos a rede neural que tem o papel de criar as relações entre as palavras, classificando-a como algo bom, ruim, não tão bom, não tão ruim, neutro, etc.

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=len(word_index) + 1, embedding_dim=16),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Ao realizar o treinamento do modelo, foi possível perceber que provavelmente estamos lidando com overfitting, visto que o modelo tem valor de perda nos dados de treino diminuindo a cada iteração, já os dados de validação só aumentam com o passar das épocas. Pode-se verificar também que até a acurácia na validação começou a cair.



Para testar o modelo, utilizei cinco frases descritas abaixo, onde as com coloração verde são sarcásticas:

- "granny starting to fear spiders in the garden might be real"
- "oh, don't worry, I'm sure that plan will fail spectacularly"
- "game of thrones season finale showing this sunday night"
- "oh, fantastic idea. Let's ignore reality and go with that"
- "wow, you must be exhausted from all that thinking you didn't do"

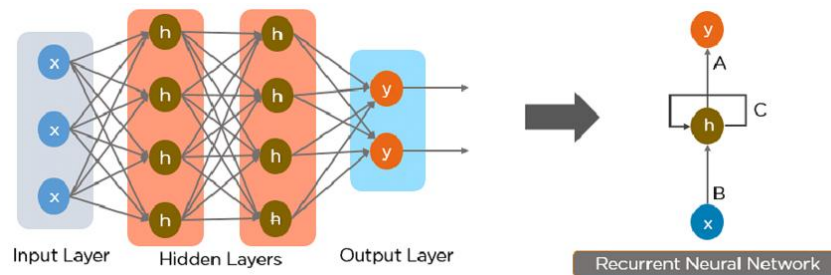
Podemos verificar na imagem abaixo que, apenas três frases foram classificadas corretamente, a 2ª, 3ª (como ela não é sarcástica, o valor está muito baixo) e 5ª frase. A primeira existe a dúvida, visto que o valor é baixo, mas não como a classificação da 3ª frase, já a 4ª frase foi classificada de longe como não sarcástica, o que não corresponde com a resposta que esperávamos.

```
[[3.0447431e-03]
 [9.999738e-01]
 [2.3385147e-07]
 [1.4121404e-06]
 [1.0000000e+00]]
```

Redes Neurais Recorrentes (RNNs)

As Redes Neurais Recorrentes (RNNs) são uma extensão das ANNs, projetadas especificamente para lidar com dados sequenciais, como séries temporais e linguagem natural. Diferenciam-se por manterem uma memória dos estados anteriores por meio de conexões recorrentes, o que as torna ideais para tarefas como tradução automática, geração de texto, análise de sentimentos e previsão de séries temporais.

A ideia é passar o contexto das palavras para o próximo neurônio da rede, tornando a rede como se fosse um banco de informações sobre uma determinada frase. Existe também algo chamado “Cell State” que fará um papel importante de manter as informações e sentidos da frase.

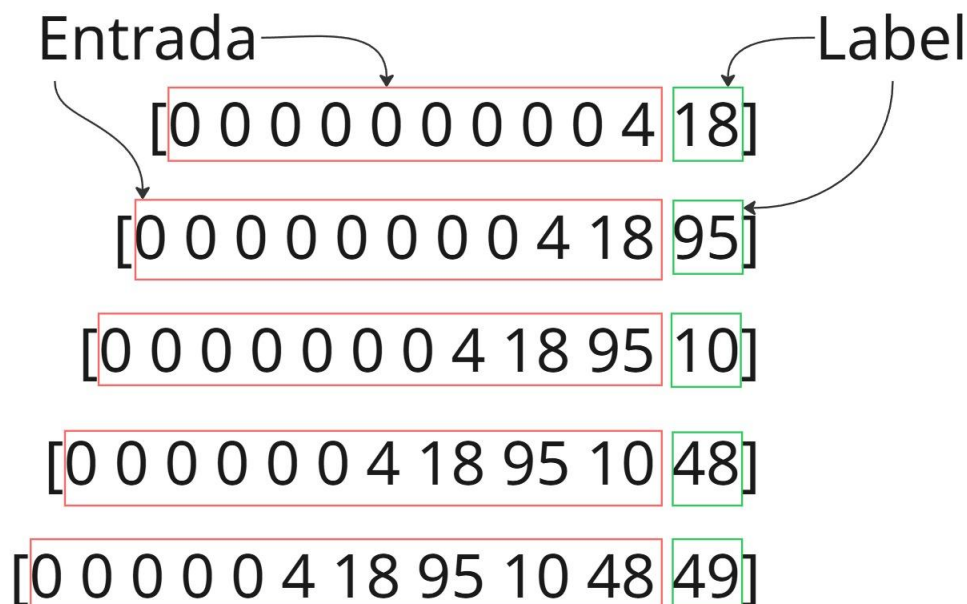


No exemplo de aula, utilizamos um poema para treinar a nossa rede neural recorrente para tentar completar as próximas palavras de uma determinada frase. <<https://storage.googleapis.com/learning-datasets/irish-lyrics-eof.txt>>

Vale ressaltar que aqui não dividiremos os dados (sentenças) em treino e teste, usaremos tudo que temos para treinar o modelo e deixa-lo da melhor forma possível. O primeiro passo foi tokenizar as palavras do poema, que totalizaram 2690 palavras únicas.

Para que o modelo consiga prever uma palavra que será preenchida, teremos de fazer um treinamento específico, que consiste da seguinte forma:

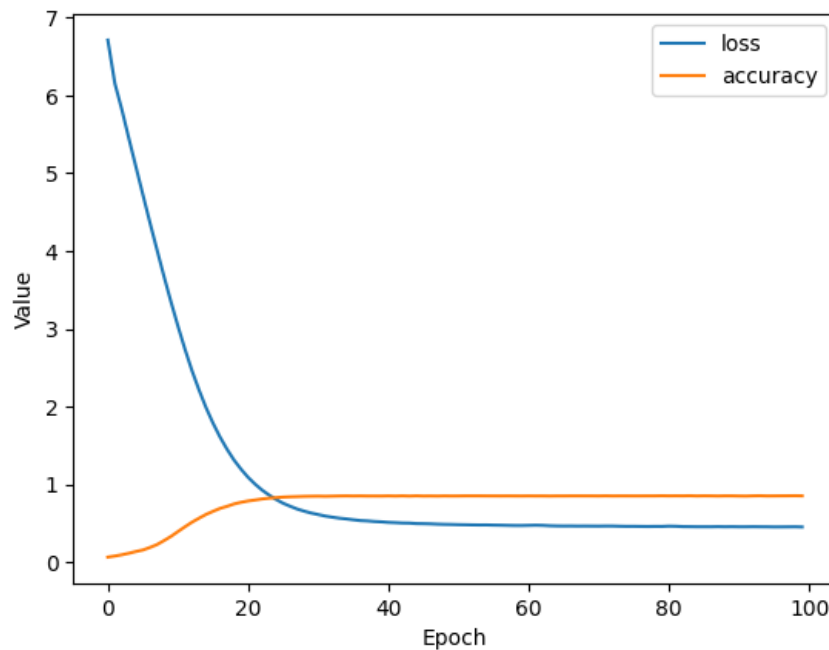
Passaremos para o modelo a entrada, que seriam palavras, e a label que consista na “resposta”, indicando que após aquela palavra ou frase recebida na entrada, ao final dela virá essa palavra contida na label. Note que as frases já estão com o padding aplicado.



É importante salientar também que a label é codificada no tipo One-Hot Encoding, em que cada palavra tem um ID único em um vetor de 0s e 1s, onde o valor 1 representa uma palavra no total de palavras presentes.

Passamos isso então para uma RNN, com camada Embedding de entrada, em seguida uma camada Bidirecional que armazenará o contexto e informações entre palavras e por fim uma camada Densa com o total de palavras (2690).

Temos o seguinte resultado obtido pelo modelo, onde os valores de loss e acurácia chegaram em um plateau e não obtiveram melhora.



Realizando uma previsão com o modelo, limitando-o a prever 20 palavras a partir da frase "Walking on a dream", obtivemos o seguinte resultado:

```
1/1 ————— 0s 52ms/step
1/1 ————— 0s 39ms/step
1/1 ————— 0s 39ms/step
1/1 ————— 0s 38ms/step
1/1 ————— 0s 38ms/step
1/1 ————— 0s 38ms/step
1/1 ————— 0s 38ms/step
1/1 ————— 0s 38ms/step
1/1 ————— 0s 38ms/step
1/1 ————— 0s 38ms/step
1/1 ————— 0s 38ms/step
1/1 ————— 0s 45ms/step
1/1 ————— 0s 47ms/step
1/1 ————— 0s 37ms/step
1/1 ————— 0s 47ms/step
1/1 ————— 0s 36ms/step
1/1 ————— 0s 43ms/step
1/1 ————— 0s 43ms/step
1/1 ————— 0s 37ms/step
1/1 ————— 0s 37ms/step
Walking on a dream by the river my love and i did stand your eyes eyes eyes eyes eyes eyes eyes eyes eyes
```

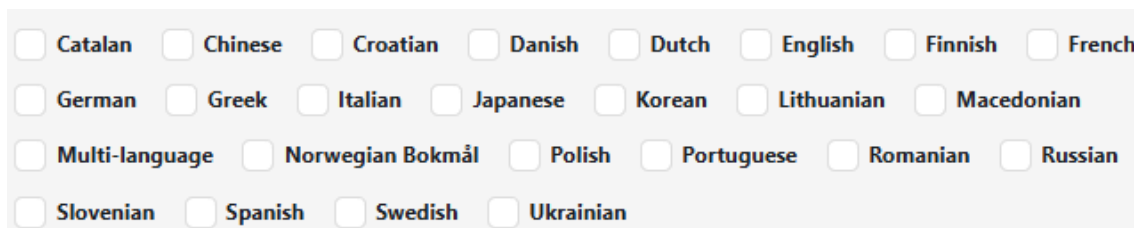
É possível verificar que ele até conseguiu gerar algumas palavras que fizessem sentido, mas ao chegar na 11ª palavra, ele começou a repetir apenas "eyes". Talvez seja possível corrigir essa limitação melhorando a rede neural e talvez aumentar os dados.

Natural Language Processing with spaCy & Python - Course for Beginners

Essa aula foi extremamente interessante, dado que foi apresentado a biblioteca spaCy para utilizar no processamento de linguagem natural.

É importante salientar que o spaCy é algo além de codificar palavras com números, como citado na seção anterior, ele é muito mais complexo e permite manter o sentido, dependências entre palavras, onde a palavra começa e termina, enfim, possui inúmeras funcionalidades que serão imprescindíveis no cenário aplicado (NLP).

O primeiro ponto que me impressionou é a facilidade com que um texto/tsv é carregado e interpretado pelo spaCy, basta possuir um arquivo de texto e carrega-lo no Pyhon, isso claro, levando em consideração um cenário em que não existem erros de digitação, siglas, sentido, etc. Essa biblioteca tem modelos diferentes para cada língua, abrangendo as seguintes:



Em um dos exemplos de aula, o autor carregou um texto da Wikipedia. Para carregar esse texto no spaCy, existe um tipo fundamental que é chamado de “Doc Container”, ele que fará o gerenciamento de todas as sentenças do texto.

Aplicando o modelo NLP do spaCy, as palavras foram automaticamente transformadas para tokens, mas mais que isso, a biblioteca criou como se fosse um array de características, dependências, tipo, relação com outras palavras no texto, dentre outros, para cada palavra. Na Figura abaixo conseguimos analisar essas dependências criadas.

Os atributos de tokens presentes no spaCy estão listados abaixo. Cada token presente no texto terá esses atributos, permitindo o usuário realizar uma análise muito detalhada (vale notar que essa análise poderá se tornar útil quando quisermos filtrar um texto utilizando qualquer atributo abaixo).

- .text
- .head
- .left_edge
- .right_edge
- .ent_type_
- .iob_
- .lemma_
- .morph
- .pos_
- .dep_
- .lang_

```
for tok in doc[0:20]:
    print(tok.text, tok.pos_, tok.dep_, tok.head.text, tok.head.pos_)
# .text - texto do token
# .pos_ - parte do discurso (POS)
# .dep_ - dependência sintática
# .head - cabeça do token (palavra que governa este token)
# .head.text - texto da cabeça do token
# .head.pos_ - parte do discurso da cabeça do token
```

```
The DET det States PROPN
United PROPN compound States PROPN
States PROPN nsubj is AUX
of ADP prep States PROPN
America PROPN pobj of ADP
```

Isso contorna um problema que, por exemplo, era de buscar por um nome próprio (PROPN) e um verbo (VERB), em sequência. A ideia de tokenizar com números não tinha essa relação, muito menos que tipo era a palavra (verbo, substantivo, adjetivo), para o computador eram apenas números que ocorriam em um determinado padrão e tinham certa relação.

A biblioteca tem inúmeras utilidades textuais, que permitem o usuário analisar de diferentes formas um texto. A primeira são as sentenças, que basicamente divide o texto quando há a presença de um “.”. É interessante salientar que o modelo carregado consegue discernir se um “.” está entre parênteses, aspas, dentre outros, não representando realmente o fim de uma sentença. A Figura abaixo demonstra esse caso, onde existe “.”s que não indicam o fim de frase.

```
for sent in doc.sents:
    print(sent.text)

The United States of America (U.S.A. or USA), commonly known as the United States (U.S. or US) or America, is a country primarily located in North America. It consists of 50 states, a federal district, five major unincorporated territories, 326 Indian reservations, and some minor possessions.[j] At 3.8 million square miles (9.8 million square kilometers), it is the world's third- or fourth-largest country by total area.[d]
```

Word Vectors

Word Vectors é uma forma de representação do texto, em que é feito com números inteiros únicos, tokenizando-os com 1, 2, 3, etc. Isso é utilizado para verificar a similaridade entre um texto e outro, ou similaridade de palavras. A Figura abaixo permite analisar esse cenário.

```
doc1 = nlp("I like salty fries and hamburguers.")
doc3 = nlp("Fast Food tastes very good.")
```

```
print(doc1, "<=>", doc3, doc1.similarity(doc3))
```

```
I like salty fries and hamburguers. <=> Fast Food tastes very good. 0.772051215171814
```

É possível também quando queremos buscar sinônimos ou palavras similares a uma determinada palavra. O caso abaixo mostra esse cenário, onde uma palavra é definida e então o objetivo é procurar palavras similares ou com sentido parecido a essa.

```
your_word = "dog"

ms = nlp.vocab.vectors.most_similar(
    np.asarray([nlp.vocab.vectors[nlp.vocab.strings[your_word]]]), n=10)
words = [nlp.vocab.strings[w] for w in ms[0][0]]
distances = ms[2]
print(words)
```

```
['dog', 'KENNEL', 'dogs', 'CANINES', 'GREYHOUND', 'pet', 'Pet-Care', 'FELINE', 'cat', 'BEAGLES']
```

Assim como podemos fazer com um texto, podemos analisar palavras também.

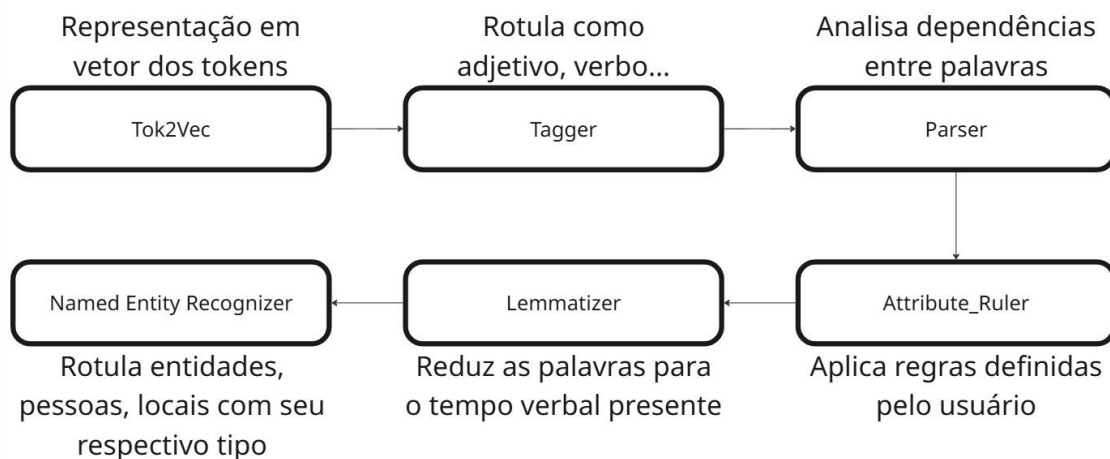
```
print(doc5[2], "<=>", doc6[2:5], doc5[2].similarity(doc6[2:5]))
```

```
trucks <=> semi-trucks 0.6570225954055786
```

Custom Pipelines

O spaCy possui um pipeline padrão definido pelo modelo que estamos utilizando. O modelo `small` (`en_core_web_sm`) possui um pipeline composto pelas seguintes etapas:

Pipeline do spaCy



Podemos também inserir o nosso próprio pipeline a partir de um modelo em branco, ou também remover e adicionar etapas de um modelo já criado, como o `en_core_web_sm`.

É interessante essa personalização, visto que o usuário pode querer formatar certos conjuntos de palavras de uma determinada forma, como é o caso abaixo, onde uma suposta cidade chamada `West Chestertenfieldville` é rotulada como GPE (Geopolitical Entity).


```
patterns = [
    {"label": "GPE",
     "pattern": "West Chestertenfieldville"}
]
```

Note que ela não está adicionada ao pipeline, é necessário que você a insira um Entity_Ruler no pipeline, e assim inserir os patterns adicionados pelo usuário.

Ainda nesse contexto do Custom Pipelines, podemos também adicionar componentes (como parser, tok2vec, tagger) customizados. Por exemplo, é possível que adicionemos um componente para remover tokens que foram rotulados como GPE.

```
@Language.component("remove_gpe")
def remove_gpe(doc):
    ents = list(doc.ents)
    for ent in ents:
        if ent.label_ == "GPE":
            ents.remove(ent)
    doc.ents = ents
    return doc
```

Ao adicionar esse componente ao pipeline, ele estará disposto ao final dele por padrão. Existem alguns componentes que serão necessários adicionar antes ou depois de alguma etapa já presente no pipeline. Nesse caso é importante que o pipeline tenha sido completo por conta de ser uma exclusão por rótulo.

Matcher

O Matcher, de forma simples, consiste em uma forma de você buscar informações em um determinado texto. Mais que isso, ele permite analisar padrões, formato da palavra, tipo de palavra, se é um número, dígito, se é um URL, e-mail, enfim, permite muitos tipos de pesquisas em texto.

Ela se torna uma ferramenta muito poderosa quando combinamos vários padrões que queremos buscar, como é o caso abaixo. Na variável "pattern" nós temos o que deverá ser buscado pelo Matcher. O "POS" indica que iremos buscar por um "Part of speech" do tipo PROPN, que indica um proper name (nome próprio). O "OP" é um indicador de quantidade, onde o "+" representa um ou mais. Note a utilização do parâmetro "greedy=LONGEST", ele fará com que o Matcher não quebre os nomes, ou seja, não o divida em partes menores, mas sim pegar a maior sequência encontrada.

```
matcher = Matcher(nlp.vocab)
pattern = [{"POS": "PROPN", "OP": "+"}]
matcher.add("PROPER_NOUN_PATTERN", [pattern], greedy="LONGEST")
doc = nlp(text)
matches = matcher(doc)
print(matches.__len__())
for match in matches[:10]:
    print(match, doc[match[1]:match[2]])
```

Conseguimos analisar o seguinte resultado:

```
(54612916926452064, 0, 4) Martin Luther King Jr.  
(54612916926452064, 6, 9) Michael King Jr.  
(54612916926452064, 10, 11) January  
(54612916926452064, 16, 17) April  
(54612916926452064, 50, 51) King  
(54612916926452064, 70, 72) Mahatma Gandhi  
(54612916926452064, 84, 89) Martin Luther King Sr.  
(54612916926452064, 90, 91) King  
(54612916926452064, 114, 115) King  
(54612916926452064, 118, 119) Montgomery
```

Claro que existem alguns erros, como nomes de meses do ano. Isso se dá por conta de como isso foi escrito no texto, tornando-o um nome “próprio”.

Por fim, existem os inúmeros atributos que podemos personalizar para fazer uma busca em texto, eles estão listados abaixo. Cada um deles devem ter a estrutura mostrada acima, ou seja, seria algo assim: “IS_ALPHA” : True; “POS” : “VERB”.

- ORTH
- TEXT
- LOWER
- LENGTH
- IS_ALPHA
- IS_ASCII
- IS_DIGIT
- IS_LOWER
- IS_UPPER
- IS_TITLE
- IS_PUNCT
- IS_SPACE
- IS_STOP
- IS_SENT_START
- LIKE_NUM
- LIKE_URL
- LIKE_EMAIL
- SPACY
- POS
- TAG
- MORPH
- DEP
- LEMMA
- SHAPE
- ENT_TYPE
- _ - Custom extension attributes (Dict[str, Any])
- OP

Note também que os patterns são sequenciais, então você precisa monta-los na ordem que deseja, como extrair um texto que começa com um PROPON, em seguida um VERB, e por fim uma vírgula seguida de um texto entre aspas.

Regex no spaCy

Esse tópico me fez lembrar muito da disciplina de Compiladores, onde no curso utilizamos expressões regulares para encontrar sequências de caracteres em um texto. A expressão regular abaixo representa um exemplo aplicado, onde buscamos por o nome “Paul”, seguido de qualquer sequência de letras onde a primeira é maiúscula seguida por qualquer caractere alfanumérico.

```
pattern = r"Paul [A-Z]\w+"
```

Aplicando essa expressão regular acima em um texto qualquer, é possível verificar que foi encontrado correspondências com o que desejamos.

```
matches = re.finditer(pattern, text)

for match in matches:
    print(match)

<re.Match object; span=(0, 11), match='Paul Newman'>
<re.Match object; span=(39, 53), match='Paul Hollywood'>
```

Podemos também criar um componente que irá ser inserido dentro de um pipeline para fazer isso automaticamente para nós, ou seja, o spaCy possui muitas possibilidades de utilização dos seus recursos.

Podemos fazer muitas combinações de processos que almejamos fazer, como criar um Matcher que irá buscar por sequências de Entidades Geopolíticas (GPE) que possuem um verbo em sequência, em seguida passar isso para um componente, onde será analisado automaticamente ao passar para o pipeline e passado a uma camada seguinte.

Outra opção que podemos realizar é extrair entidades (ex: pessoas) e verificar suas relações sintáticas (ex: sujeito → verbo → objeto) para mapear "quem fez o quê".

Referencias

[Playlist Natural Language Processing - Tokenization \(NLP Zero to Hero\)](#)

[Natural Language Processing with spaCy & Python - Course for Beginners](#)