**Corretor Ortográfico em Python – Aplicando técnicas de NLP**

Link do google drive:

<https://drive.google.com/drive/folders/1wnumjuGByvZKLdVXVk39JLhO5RNyNUdt>.

Link das libs:

- <https://www.nltk.org/>

1. **Aula 1 – Explorando Um Projeto De NLP:**
   1. NLP: do inglês significa processamento de linguagem natural. Responsável por fazer o intermédio da comunicação entre humanos e máquinas, tendo diversas aplicações como o google assistente, siri, alexa dentre outras possibilidades como análise de sentimento, tradutores, buscadores e corretores ortográficos.
   2. Importando Um Corpus Textual:
      1. Precisamos ter uma base de dados para que o nosso algoritmo conheça as palavras para que possa corrigir.
      2. Quando trabalhamos NLP a nossa base de dados se chama corpus, sendo um conjunto de documentos e, para ler um arquivo de texto em python precisamos fazer um código simples, onde dizemos que com o nosso arquivo aberto em formato de leitura, como f, atribua a leitura de f em artigos:

with open('artigos.txt', 'r') as f:

  artigos = f.read()

* + 1. Assim, quando printarmos artigos, todo o nosso corpus será exibido.
  1. Tokenização:
     1. A primeira coisa para saber se nosso corpus é ideal, é saber a quantidade de palavras que ele possui, além de saber se elas se repetem muito ou não.
     2. Tokens são pequenos pedaços do nosso corpus composto por palavras e pontuações, uma vez que quando utilizamos o método split() para quebrar todo o nosso texto em uma lista com palavras, ele não separa somente as palavras, mas a concatenação delas com a pontuação também:

texto\_exemplo = 'Olá, tudo bem?'

token = texto\_exemplo.split()

print(palavras\_separadas)



* 1. O que aprendemos:
     1. O que é o processamento de linguagem natural (NLP);
     2. O que está por trás de um corretor ortográfico;
     3. Como ler um arquivo de texto utilizando python;
     4. Como contar a quantidade de caracteres de um texto usando len().

1. **Aula 2 – Utilizando NLTK Para Tokenizar Um Texto:**
   1. Refinando a Tokenização: natural linguage toolkit
      1. Importamos e fizemos a instalação do punkt para podermos utilizar o método nltk.tokenize.word\_tokenize(texto\_exemplo) separando as palavras da pontuação:

import nltk

nltk.download('punkt')

palavras\_separadas = nltk.tokenize.word\_tokenize(texto\_exemplo)

palavras\_separadas



* 1. Separando Palavras De Tokens:
     1. Podemos criar uma função para separar fazendo um for que percorra toda a nossa lista de tokens e, se a palavra atual Isalpha(), ou seja, for alfabética, não contendo números, pontuação e afins, então adiciona a uma nova lista, retornando ela no final:

def separa\_palavras(lista\_tokens: list):

  lista\_palavras = []

  for token in lista\_tokens:

    if token.isalpha():

      lista\_palavras.append(token)

  return lista\_palavras

separa\_palavras(palavras\_separadas)



* 1. Contando Palavras Do Corpus:
     1. Fizemos exatamente a mesma coisa que acima, mas com o nosso artigo, e fizemos a contagem da quantidade de palavras. Podemos ver que temos um número BEM significativo de palavras no nosso corpus:

lista\_tokens = nltk.tokenize.word\_tokenize(artigos)

lista\_palavras = separa\_palavras(lista\_tokens)

print(f'O número de palavras é: {len(lista\_palavras)}')



* 1. Normalização:
     1. Apesar de termos essa quantidade gigante de palavras no nosso corpus, sabemos que na realidade não temos tudo isso de palavras diferentes, uma vez que em um único texto repetimos a mesma palavra várias vezes. Considerando isso precisamos fazer a normalização do nosso corpus, mantendo somente palavras diferentes.
     2. A normalização consiste em manter somente 1 tipo de cada palavra, ou seja, escrita de uma única forma e apenas uma vez, por exemplo: ‘Temos’ e ‘temos’. São considerados 2 tipos pois começam com letras maiúscula e minúscula. Para fazer essa normalização criamos uma função que acessa cada palavra da nossa lista e adiciona ela em outra lista, mas com todas as letras minúsculas, ou seja, outro for:

def normalizacao(lista\_palavras: list):

  lista\_normalizada = []

  for palavra in lista\_palavras:

    lista\_normalizada.append(palavra.lower())

  return lista\_normalizada

* 1. Tipos De Palavras:
     1. Para remover as repetições de uma lista utilizamos o set(). Após fazer isso com nossa lista e verificarmos a quantidade de tipos que temos, podemos ver que o número diminui consideravelmente, indo de 393k para 17k. Contudo, após uma pesquisa, vimos que a quantidade de palavras necessárias para uma pessoa ser considerada fluente é de 8k, ou seja, 17k é mais do que o suficiente:

lista\_unica = set(lista\_normalizada)

len(lista\_unica)



* 1. O que aprendemos:
     1. O que são tokens;
     2. O que é e como utilizar a biblioteca NLTK;
     3. Como tokenizar um texto usando NLTK;
     4. Como separa apenas caracteres alfabético de um token;
     5. O que são tipos de palavras.

1. **Aula 3 – Desenvolvendo e Testando o Corretor:**
   1. Detalhando o Corretor:
      1. O algoritmo que vamos fazer consiste de separar a nossa palavra digitada em 2 partes, realizar uma operação e retornar um resultado. Basicamente ele vai pegar o índice 0, ou seja, antes da primeira letra da nossa palavra e testar todas as letras, tendo como resultado uma palavra com uma nova letra no início, repetindo esse processo para todos os índices da palavra até o resultado bater com o banco de dados de palavras:



* 1. Fatiando Strings:
     1. Utilizamos fatiamento para fazer a separação do lado direito para o esquerdo, criando uma tupla para cada caso e colocando eles em uma lista. Para isso fizemos um for onde o index, no tamanho da palavra + 1 (para que a última tupla seja a palavra toda na esquerda e nada na direita), a lista fatias adiciona o fatiamento no index atual:

palavra\_exemplo = 'lgica'

def gerador\_palavras(palavra):

  fatias = []

  for i in range(len(palavra) + 1):

    fatias.append((lista[:i],lista[i:]))

  print(fatias)

gerador\_palavras(palavra\_exemplo)



* 1. Operação De Inserção:
     1. Criamos uma função que para cada lado direito e esquerdo da nossa fatia e para cada letra na nossa lista de letras possíveis no nosso alfabeto, faz a interpolação do lado esquerdo com a letra e com o lado direito, adicionando e retornando a nossa lista vazia de novas palavras. Colocamos essa função interligada com o nosso gerador de palavras:

palavra\_exemplo = 'lgica'

def insere\_letras(fatias):

  novas\_palavras = []

  letras = 'abcdefghijklmnopqrstuvwxyzàáâãèéêìíîòóôõùúûç'

  for E, D in fatias:

    for letra in letras:

      novas\_palavras.append(E + letra + D)

  return novas\_palavras

def gerador\_palavras(palavra):

  fatias = []

  for i in range(len(palavra) + 1):

    fatias.append((palavra[:i],palavra[i:]))

  palavras\_geradas = insere\_letras(fatias)

  return palavras\_geradas

palavras\_geradas = gerador\_palavras(palavra\_exemplo)

palavras\_geradas

  ...

* 1. Construindo a Função Corretor:
     1. Demos início passando como parâmetro a palavra que queremos corrigir. Nossa função tem uma variável que recebe o gerador de palavras que gera as palavras a partir da palavra errada. Temos uma segunda variável que é a que recebe a palavra correta através da função max() que retorna o valor de maior probabilidade através da chave probabilidade, que é outra função que não criamos ainda. No final ela retorna a palavra certa:

def corretor(palavra):

  palavras\_geradas = gerador\_palavras(palavra)

  palavra\_correta = max(palavras\_geradas, key = probabilidade)

  return palavra\_correta

* 1. Probabilidade Das Palavras Geradas:
     1. O primeiro passo é calcular a probabilidade delas a partir da divisão da frequência da palavra dividido pelo total de palavras.
     2. O nltk já nos ajuda a fazer isso com o nltk.FreqDist(lista\_normalizada), nos devolvendo uma lista de tuplas da palavra e sua frequência:

frequencia = nltk.FreqDist(lista\_normalizada)

frequencia["lógica"]



* + - 1. A partir disso conseguimos construir nossa função de probabilidade:

def probabilidade(palavras\_geradas):

  return frequencia[palavras\_geradas]/total\_palavras

* + 1. A partir disso nosso corretor começou a funcionar adequadamente:

corretor('lgica')



* + 1. Apesar de ter funcionado corretamente, nosso corretor só corrige esse tipo de erro, por enquanto. Vamos incrementar na próxima aula.
  1. O que aprendemos:
     1. Um algoritmo capaz de corrigir palavras com uma letra a menos;
     2. Como implementar este algoritmo;
     3. Como utilizar estatística para selecionar a palavra correta;
     4. Como calcular a distribuição de frequência de das palavras de com corpus textual.

1. **Aula 4 – Avaliando a Qualidade do Corretor:**
   1. Preparando Dados de Teste:
      1. Fizemos uma função que recebe como parâmetro um arquivo txt de palavras onde, em cada linha, temos a palavra escrita de maneira correta na esquerda e de maneira errada na direita. Nossa função então cria uma lista vazia e em seguida abre nosso arquivo txt em formato de leitura, atribuindo a variável f (como fizemos lá no início de tudo) e faz um for onde para cada linha de f (nosso arquivo de leitura) separa em duas variáveis ‘certo’ errado’ a partir de um split() da linha atual. Em seguida a nossa lista vazia vai receber um append() de uma tupla, onde estão as palavras correta e errada. Depois do for fechamos o txt e retornamos a nossa lista de palavras com as tuplas:

def cria\_dados\_teste(nome\_arquivo):

  lista\_palavras\_teste = []

  f = open(nome\_arquivo, 'r')

  for linha in f:

    correta, errada = linha.split()

    lista\_palavras\_teste.append((correta, errada))

  f.close()

  return lista\_palavras\_teste

cria\_dados\_teste('palavras.txt')



* 1. Avaliando o Corretor:
     1. Criamos a função avaliador que recebe nossa lista de palavras teste como parâmetro. Para saber qual é a taxa de acerto precisamos dividir a quantidade de acertos pelo total de palavras, recebendo a porcentagem de quantas palavras foram acertadas. Considerando isso, criamos uma variável que recebe o tamanho da nossa lista de palavras testes e outra que recebe a quantidade de acertos = 0, uma vez que iniciamos o código com 0 acertos. Em seguida precisamos criar um for para iterar nossa lista de palavras teste (a lista criada acima), onde separa as tuplas em duas variáveis: certas e erradas (também como feito acima). Fazemos essa distinção pois vamos, dentro desse ‘for’, criar uma variável que recebe a palavra corrigida pelo nosso corretor e uma verificação, que diz que, se a palavra corrigida for igual a palavra certa da tupla da nossa lista de palavras teste, então a variável acertou recebe ela + 1, contando mais 1 acerto para fazer o cálculo da taxa de acertos logo após o for, em seguida printando qual foi a porcentagem de acertos do total de palavras da nossa lista de palavras teste:

def avaliador(testes):

  numero\_palavras = len(testes)

  acertou = 0

  for correta, errada in testes:

    palavra\_corrigida = corretor(errada)

    if palavra\_corrigida == correta:

      acertou += 1

  taxa\_acerto = acertou/numero\_palavras

  print(f'Taxa de acerto: {taxa\_acerto\*100:.2f}% de {numero\_palavras} palavras')

avaliador(lista\_teste)



* 1. O que aprendemos:
     1. O que é preciso para avaliar um corretor;
     2. Como estruturar os dados de teste para avaliação do corretor;
     3. Calcular a taxa de sucesso do corretor.