**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')

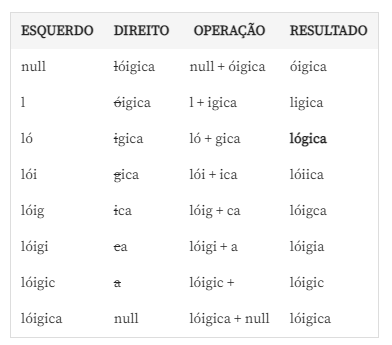
avaliador(lista\_teste)



* 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.

1. **Aula 5 – Incrementando o Corretor:**
   1. Operação de Delete:
      1. O que vamos fazer é parecido com a lógica de adicionar um caracter. Precisamos dividir a palavra em esquerdo e direito, onde o direito vai pegar sempre um caracter pra frente do seu início, ou seja, na palavra ‘lóigica’ ele irá separar em ‘l’ e ‘igica’, eleminando o ‘ó’ da palavra por ter pego da segunda letra do lado direito pra frente. Precisaremos que nosso algoritmo faça isso para todas as posições da nossa palavra, assim como na incrementação de letras. Segue uma tabela de exemplo de como a lógica será feita, como colocamos para a correção de letras faltantes:





* 1. Implementando o Delete de Caracteres:
     1. Como vimos no tópico acima, o código é bem similar para adicionar a nova palavra, então o que fizemos foi copiar o código que adiciona e fizemos uma adaptação:

def deletando\_caracter(fatias):

  novas\_palavras = []

  for E, D in fatias:

    novas\_palavras.append(E + D[1:])

  return novas\_palavras

* 1. Avaliando o Novo Corretor:
     1. Fizemos uma modificação no nosso gerador de palavras, onde a nossa lista de palavras geradas, além de receber as palavras da função insere\_letras() também recebe a do deleta\_caracter(), dessa forma fazendo com que o corretor funcione corretamente:

def gerador\_palavras(palavra):

  fatias = []

  for i in range(len(palavra) + 1):

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

  palavras\_geradas = insere\_letras(fatias)

  palavras\_geradas += deletando\_caracter(fatias)

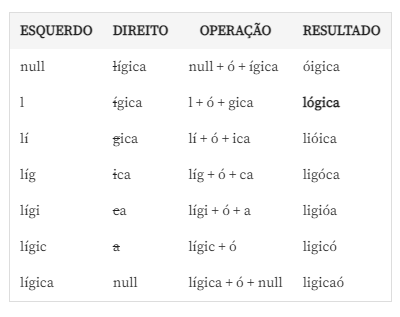
  return palavras\_geradas

* + 1. Ao avaliar novamente nosso corretor através da lista teste que temos, vemos que agora a nossa taxa de acerto subiu para 41.40% de 186 palavras.
  1. O que aprendemos:
     1. Um novo algoritmo para melhorar o corretor;
     2. Como criar uma função para deletar caracteres de uma palavra incorreta.

1. **Aula 6 – Corrigindo os Principais Erros de Digitação:**
   1. Trocando as Letras:
      1. A lógica dessa correção junta as duas anteriores, onde nosso corretor deve separar nossa palavra em 2 partes, onde a parte direita é deletando o primeiro caracter, e em seguida concatenar o lado esquerdo com uma letra e depois com o lado direito, ou seja:







* 1. Implementando a Troca de Letras:
     1. Como disse acima, basicamente só precisamos colocar o índice 1 na hora de concatenar com o lado direito na função insere letras, ficando assim:

def troca\_letra(fatias):

  novas\_palavras = []

  letras = 'abcdefghijklmnopqrstuvwxyzàáâãèéêìíîòóôõùúûç'

  for E, D in fatias:

    for letra in letras:

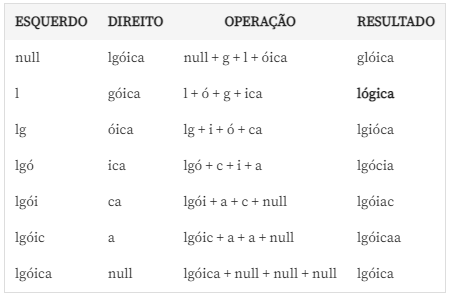
      novas\_palavras.append(E + letra + D[1:])

  return novas\_palavras

* 1. Invertendo as Letras:
     1. Um erro bem comum quando estamos digitando é inverter 2 letras. O algoritmo que vamos criar para corrigir isso consiste em novamente fatiar em 2 partes, inverter a primeira letra do lado direito com a segunda, também do lado direito, e concatenar tudo:







* 1. Implementando a Inversão de Letras:
     1. Basicamente o que precisamos fazer é copiar a função que deleta caracter e fazer a alteração da operação de concatenação, mas com a condição de que o tamanho do lado direito do nosso fatiamento seja maior que 1, pois, como vimos na tabela acima, se não tiver nada no lado direito ou apenas 1 caracter, não tem como fazer inversão:

def inverte\_letra(fatias):

  novas\_palavras = []

  for E, D in fatias:

    if len(D) > 1:

      novas\_palavras.append(E + D[1] + D[0] + D[2:])

  return novas\_palavras

* + 1. Lembrando, é claro, de sempre adicionar essa nova função no gerador de palavras:

def gerador\_palavras(palavra):

  fatias = []

  for i in range(len(palavra) + 1):

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

  palavras\_geradas = insere\_letras(fatias)

  palavras\_geradas += deletando\_caracter(fatias)

  palavras\_geradas += troca\_letra(fatias)

  palavras\_geradas += inverte\_letra(fatias)

  return palavras\_geradas

* + 1. Ao avaliar o nosso corretor, vemos que nossa taxa de acerto está em quase 76%, aumentando consideravelmente.
    2. Esses 24% de erro pode estar acontecendo por 2 motivos: falta implementação de correções ou falta de conhecimento/vocabulário por parte do nosso algoritmo. Vamos ver como saber qual a porcentagem de vocabulário do nosso algoritmo na próxima aula.
  1. O que aprendemos:
     1. Quais são os principais erros de digitação;
     2. Os algoritmos para trocar e inverter letras de uma palavra.

1. **Aula 7 – Criando um Corretor Turbinado:**
   1. Palavras Desconhecidas Ao Vocabulário:
      1. Para gerar a taxa de palavras desconhecidas também vamos utilizar a função avaliador(). O próprio código é bem parecido, uma vez que o que fizemos foi criar um contador para as desconhecidas e implementar um else onde essa variável vai receber ela mesmo + a palavra correta caso ela não esteja no nosso vocabulário, sendo esse o set() da nossa lista normalizada:

def avaliador(testes, vocabulario):

  numero\_palavras = len(testes)

  acertou = 0

  desconhecida = 0

  for correta, errada in testes:

    palavra\_corrigida = corretor(errada)

    if palavra\_corrigida == correta:

      acertou += 1

    else:

      desconhecida += (correta not in vocabulario)

  taxa\_acerto = acertou/numero\_palavras

  taxa\_desconhecida = desconhecida/numero\_palavras

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

  print(f'Taxa de desconhecidas: {taxa\_desconhecida\*100:.2f}%')

avaliador(lista\_teste, vocabulario)



* 1. Turbinando O Gerador De Palavras:
     1. Atualmente nosso corretor não é capaz de corrigir palavras que necessitem de 2 operações ou mais para sua correção, ou seja, só consegue corrigir palavras que estão a 1 passo da palavra certa.
     2. Para fazer essa correção e permitir que nosso corretor corrija palavras que estão a 2 operações de distância da palavra correta, criamos outra função que recebe as palavras geradas pelo gerador de palavras e, para cada palavra dessa lista, ele faz passar novamente pelo gerador criando uma lista de novas palavras e retornando elas:

def gerador\_turbinado(palavras\_geradas):

  novas\_palavras = []

  for palavra in palavras\_geradas:

    novas\_palavras += gerador\_palavras(palavra)

  return novas\_palavras

* + 1. Apesar de o resultado final ser correto, o volume de palavras que está sendo criada para a correção de uma única palavra é MUITO grande (cerca de 690K). Isso faz com que nosso algoritmo seja pouco eficiente, ficando lento. Para corrigir precisamos fazer a seleção dos melhores candidatos sem que gere tantas palavras.