Projeto Final de NLP - Autocomplete

Bruno Domingues e Michel Moraes

June 23, 2021

1 Abstract

Nos dias de hoje, quase todos os textos que escrevemos no cotidiano, são digitados em alguma plataforma virtual, sendo aplicativos de mensagem texto, e redes sociais, o exemplo mais comum, já que uma grande parte das pessoas manda mensagens de texto durante todo o dia, tanto para assuntos pessoais, ou profissionais, quanto como um meio de entretenimento. Relatórios e documentos extensos muitas vezes são redigidos em um computador, ou dispositivo móvel, também.

Uma ferramenta que pode agilizar bastante na hora de escrever um texto é sugestor automático de palavras, comumente visto em sites como o Google, e em teclados de dispositivos móveis. Desta forma, neste relatório se encontra uma análise de um dataset de frases, e buscamos responder se é possível fazer um sugestor automático simples com algumas ferramentas de Processamento Linguagem Natural com Python.

Palavras-chave: Processamento de linguagem natural. Autosuggest. Autocomplete. N-gramas.

2 Introdução

O uso de abreviações e atalhos na hora de digitar se encontra cada vez mais presente no dia a dia: seja ao mandar mensagens, escrever textos ou até mesmo para pesquisar na internet.

A partir deste desejo de economizar tempo, surgiram diversas ferramentas capazes de completar frases e palavras, conhecidas geralmente como autocomplete e autosuggest.

O autosuggest pode não só economizar tempo na hora de escrever grandes textos, ou rápidas mensagens, mas também diminui a quantidade de erros ortográficos, já que um bom autosuggest vai acertar nas sugestões de palavras na maioria das vezes, e essas não possuem erros ortográficos, ou de digitação.

Tendo em vista as várias vantagens deste tipo de ferramenta, optou-se por desenvolvê-la, utilizando um dataset e ferramentas de Python tanto para processar os dados quanto para montar a ferramenta.

3 Dados e pré-processamento

O primeiro passo para tentar construir um autocomplete é reunir um grande conjunto de frases. Para atingir tal objetivo, utilizou-se um dataset [1] que contém artigos de jornais diversos e textos de blogs. O dataset possui fontes de diversas línguas, e para este desenvolvimento optou-se por utilizar a língua inglesa.

Além do texto, o dataset disponibiliza também a data de publicação e a fonte do texto (geralmente um link para o artigo/blog); no entanto, estas informações não são relevantes, e podem ser descartadas.

3.1 Criando o dataframe inicial

O código abaixo mostra o processo de importar o dataset e filtrar por língua. Visto que o arquivo original era muito grande, utilizou-se a biblioteca Dask para processar o arquivo, uma vez que ela possui ferramentas e estrutura dedicada para tal.

```
# Imports utilizados
import dask.dataframe as dd
import pandas as pd
import glob, re, os

# Path do dataset
cwd = os.getcwd().replace(r"/relatorio", "")
path = os.path.join(cwd, "data")
filepath = os.path.join(path, "old-newspaper.tsv")

# Lendo os dados do arquivo tsv
ddf = dd.read_csv(filepath, sep="\t")

# Selecionando somente em inglês e somente a coluna Text
# pois os outros dados não serão utilizados
df = ddf[ddf["Language"] == "English"]
df = df[["Text"]]
```

A biblioteca Dask funciona separando o arquivo em partições diversas, o que facilita o processamento. No entanto, ao filtrar por língua foram geradas partições vazias. Como elas não serão utilizadas, elas foram removidas. Após sua remoção, converteu-se o dataframe do Dask para um dataframe do Pandas, visto que algumas chamadas de função não são as mesmas.

```
# Limpeza de particoes vazias
# https://stackoverflow.com/questions/47812785/remove-empty-partitions-in-dask
def cull_empty_partitions(df):
    ll = list(df.map_partitions(len).compute())
    df_delayed = df.to_delayed()
    df_delayed_new = list()
    pempty = None

    for ix, n in enumerate(ll):
        if 0 == n:
            pempty = df.get_partition(ix)

        else:
            df_delayed_new.append(df_delayed[ix])

    if pempty is not None:
        df = dd.from_delayed(df_delayed_new, meta=pempty)

    return df

df = cull_empty_partitions(df)
```

```
# Convertendo em um dataframe do Pandas novamente
data = df.compute()
```

O dataframe possui pouco mais de 1 milhão de artigos e blogs distintos, o que é suficiente para a análise.

3.2 Limpeza e processamento dos dados

Para poder criar a ferramenta, deve-se primeiro fazer uma limpeza no dataframe, visto que há presença de caracteres especiais, URLs e outros que podem atrapalhar a predição.

Primeiro, removeu-se todas as URLs presentes no texto.

```
# Removendo URLs
def limpa_url(texto):
    # Regex obtida de https://www.geeksforgeeks.org/python-check-url-string/
    pattern = r"""
        (?i)  # Ignore case.
        \b  # Inicio de palavra.
        (?:
            https?://
        |
            www
            \d{0,3}
            [.]
        |
            [a-z0-9.\-]+
            [.]
            [a-z]{2,4}
            /
        )
        (?:
            [^\s()<>]+
        |
            \(
            (?:
                [^\s()<>]+
            |
                \(
                [^\s()<>]+
                \)
            )*
            \)
        )+
        (?:
            \(
            (?:
                [^\s()<>]+
            |
                \(
                [^\s()<>]+
                \)
            )*
            \)
        |
            [^\s`!()\[\]{};:'\".,<>?«»“”‘’]
        )
```

```
    """
    repl = ""
    matcher = re.compile(pattern, re.VERBOSE)
    return matcher.sub(repl, texto)

data["Text"] = data["Text"].apply(lambda t: limpa_url(t))
```

Em seguida, removeu-se caracteres especiais. Vale notar que caracteres como pontos de interrogação e de exclamação devem ser conservados, pois ainda indicam o fim de uma frase, somente mudando sua entonação. Para conservá-los, substituiu-se estes caracteres pelo ponto final, visto que a entonação da frase não é o objetivo desta análise.

```
# Removendo caracteres especiais "@, #, $, %, &, *, `"
special = r"[\@\#\$\%\&\*\`]"
data["Text"] = data["Text"].apply(lambda t: re.sub(special, "", t))

# Fazendo replace de ? e ! por .
data["Text"] = data["Text"].apply(lambda t: t.replace("?", "."))
data["Text"] = data["Text"].apply(lambda t: t.replace("!", "."))
```

Em seguida, removeu-se todos os caracteres que não fossem letras, espaços, pontos finais e apóstrofes. As apóstrofes foram conservadas pois são muito utilizadas para junção de duas palavras em uma só na língua inglesa (como "don't" e "I'm").

No entanto, na hora de tokenizar estas palavras, o tokenizador as separa de forma inadequada ("don't" -> "do", "n't"). Assim, substituiu-se as apóstrofes pelo hífen, pois assim o tokenizador interpreta como uma palavra só. Esta substituição foi realizada após a remoção dos caracteres, de modo a garantir que não houvessem hífens novos que pudessem gerar problemas na hora de substituir o hífen de volta pela apóstrofe na predição.

```
# Removendo pontuações diversas exceto apostrofes e pontos finais
data["Text"] = data["Text"].apply(lambda t: re.sub(r"[^\w\s\'\.]", "", t))

# Replace de apostrofes por traços para conservar palavras
# como don't e i'm na hora de tokenizar
data["Text"] = data["Text"].apply(lambda t: t.replace("\'", "-"))
```

Após este processamento inicial, o dataframe possui frases que estão quase prontas. No entanto, como mencionado, a maior parte são artigos e textos de blogs, que possuem mais de uma frase.

Para separar as frases, utilizou-se um split usando o ponto final como caractere para definir onde uma frase acaba. Como a função split gera uma lista com as strings separadas, transformou-se esta lista em novas linhas do dataframe utilizando a função explode do Pandas.

```
# Fazendo o split das frases por .
data["Text"] = data["Text"].apply(lambda t: t.split("."))

# Comando explode para separar as listas em novas linhas do dataframe
data = data.explode("Text")
```

O dataframe agora possui apenas frases (e expandiu seu tamanho de 1 milhão de linhas para 3 milhões), pode-se continuar com a limpeza. A etapa seguinte foi a remoção de espaços no início e no fim da frase, e também a remoção de espaços excessivos.

```
# Removendo espaços em branco no início e no fim da frase
data["Text"] = data["Text"].apply(lambda t: t.strip())

# Removendo tabs, newlines e espaços em branco em excesso
data["Text"] = data["Text"].apply(lambda t: t.replace("/\s\s+/g", " "))
```

A última etapa de limpeza foi a remoção de frases que possuem números no meio. Optou-se por remover completamente essas frases, pois elas podem ser incoerentes caso sejam removidos apenas os números (por exemplo: "it costs 20 dollars" viraria "it costs dollars", o que pode não fazer muito sentido). No caso das URLs optou-se por apenas remover as URLs, pois há menos chances de ter problema de coerência.

```
# Removendo frases que contém números
def hasNumbers(text):
    return any(char.isdigit() for char in text)

data["hasNumbers"] = data["Text"].apply(lambda t: hasNumbers(t))
data = data[data["hasNumbers"] == False]
```

Por fim, removeu-se as frases vazias, filtrou-se as frases que possuem entre 7 e 12 palavras (de modo a não ter frases nem muito pequenas e nem muito grandes), e transformou-se as frases para lowercase.

```
# Removendo frases vazias
data = data[data["Text"] != ""]

# Mantendo frases que possuem entre 7 e 12 palavras
data = data[data["Text"].str.split(" ").str.len() >= 7]
data = data[data["Text"].str.split(" ").str.len() <= 12]

# Lowercase
data["Text"] = data["Text"].apply(lambda t: t.lower())

# Reset index
data.reset_index(inplace=True)

# Transforming to list
sentences = data["Text"].tolist()
```

Com as frases disponíveis, pode-se prosseguir para a construção da ferramenta.

4 Montando a ferramenta

4.1 N-gramas

Para construir a ferramenta será utilizada um modelo de N-gramas.

Um N-grama é um conceito utilizado em linguística computacional e probabilidade, e representa um conjunto de itens de um determinado texto ou fala, podendo ser sílabas, fonemas, palavras, por exemplo. Um N-grama de dois itens é chamado de bigrama, um de três itens é chamado de trigrama, etc.

Os N-gramas podem possuir algumas aplicações diferentes, e são muito utilizados nas áreas de:

- probabilidade;
- teoria da comunicação;
- linguística computacional (processamento estatístico de linguagem natural);
- biologia computacional;
- compressão de dados.

4.1.1 Modelo de predição

De modo a construir nosso preditor, utilizará-se um modelo que segue as seguintes etapas:

1. gera-se todos os N-gramas de 2 a 4 itens das frases de nosso dataframe;
2. calcula-se as probabilidades de cada N-grama estar no texto;
3. gera-se os N-gramas da nossa frase de entrada de acordo com o modelo escolhido;
4. encontra-se os primeiros N-gramas mais prováveis de serem o N-grama buscado;
5. pega-se a última palavra deles para preencher a frase de entrada.

Contudo, há palavras que são extremamente frequentes na língua inglesa (e em outras línguas). Essas são chamadas de stopwords, e para nosso modelo, os N-gramas compostos exclusivamente por stopwords serão descartados. A biblioteca NLTK possui os stopwords da língua inglesa disponível para esta remoção.

A maior vantagem do modelo de N-gramas é a sua simplicidade de entendimento e implementação. No entanto, sua maior desvantagem é que ele pode nem sempre acertar o que estava sendo buscado.

O código abaixo mostra o desenvolvimento da etapa 1 do modelo.

```
# Imports
from nltk import word_tokenize
from nltk.util import ngrams
from nltk.corpus import stopwords

# Lista dos N-gramas
bigrams = []
trigrams = []
fourgrams = []

# Stopwords da língua inglesa
stop_words = set(stopwords.words("english"))

# Iterando sobre cada frase
for sentence in sentences:
    # Tokenizando as frases
    words = word_tokenize(sentence)

    # Extendendo a lista dos N-gramas com as palavras
    bigrams.extend(list(ngrams(words, 2)))
    trigrams.extend(list(ngrams(words, 3)))
    fourgrams.extend(list(ngrams(words, 4)))
```

```python
# Remove N-gramas que são compostos exclusivamente por stopwords
def remove_stopwords(ngram: list):
    new_ngram = []

    for sequence in ngram:
        # Assume-se que o N-grama é composto por stopwords
        count = 0

        # Percorrendo cada palavra do N-grama
        for word in sequence:
            # Se o N-grama não tem nenhuma palavra que é uma stopword
            # o count vira 1; se não, se mantém em zero
            count = count or 0 if word in stop_words else count or 1

        # Adiciona-se o N-grama se ele não for composto por stopwords
        if count == 1:
            new_ngram.append(sequence)

    return new_ngram

bigrams = remove_stopwords(bigrams)
trigrams = remove_stopwords(trigrams)
fourgrams = remove_stopwords(fourgrams)
```

4.2 Probabilidades

A fórmula para calcular a probabilidade de um N-grama estar presente em um texto é dada por:

$P(N) = \frac{\text{ocorrencias(N-grama)}}{\text{total(N-gramas)}}$

Mas e se não houver ocorrências do N-grama no texto? A probabilidade neste caso seria zero. Porém, de modo a evitar estes casos, utiliza-se uma técnica de smoothing chamada Add-1 Smoothing (também conhecida como Laplace Smoothing).

Esta técnica consiste em adicionar 1 ao número de ocorrências de todos os N-gramas (ou seja, adicionar 1 ao númerador da fórmula anterior). Para compensar esta adição, devemos contar novamente quantos N-gramas temos, mas há uma forma prática de realizar isto: basta somar a quantidade de N-gramas únicos ao denominador da fórmula.

Assim, a nova probabilidade de um N-grama estar presente no texto é dada por:

$P(N) = \frac{\text{ocorrencias(N-grama)}+1}{\text{total(N-gramas)}+\text{total(N-gramas unicos)}}$

O código abaixo mostra o desenvolvimento da etapa 2 do modelo.

```python
# Utiliza-se o módulo counter para obter as contagens de cada N-grama
from collections import Counter
n2, n3, n4 = Counter(bigrams), Counter(trigrams), Counter(fourgrams)

# Gera-se os casos únicos para os bigramas, os trigramas e os quadrigramas
s2, s3, s4 = set(bigrams), set(trigrams), set(fourgrams)

# Utilizando a fórmula de probabilidades
p2 = [(n, (n2[n] + 1)/(len(n2) + len(s2))) for n in s2]
p3 = [(n, (n3[n] + 1)/(len(n3) + len(s3))) for n in s3]
p4 = [(n, (n4[n] + 1)/(len(n4) + len(s4))) for n in s4]
```

```
# Ordena-se as listas de acordo com a probabilidade de cada N-grama
# em ordem descrescente
p2 = sorted(p2, key=lambda w: w[1], reverse=True)
p3 = sorted(p3, key=lambda w: w[1], reverse=True)
p4 = sorted(p4, key=lambda w: w[1], reverse=True)
```

4.3 Gerando a predição

Para realizar a predição da próxima palavra a ser digitada em um texto, criou-se uma função que recebe como entrada:

- uma string;
- um inteiro n que define as n primeiras palavras mais prováveis a serem retornadas pelo preditor;
- um inteiro model que define o modelo de N-gramas a ser utilizado (que por padrão utiliza os bigramas);

O bloco abaixo mostra o código da função.

```
def return_prediction(inp, n, model=2):
    # Lista de possiveis palavras
    pred = []

    # Tokenizando o input
    inp = word_tokenize(inp)

    # Modelo escolhido
    p = None

    if model == 2:
        p = p2

    elif model == 3:
        p = p3

    elif model == 4:
        p = p4

    # Se a quantidade de palavras do input for menor que o tamanho do modelo
    # escolhido não é possível formar um N-grama.
    # Logo, devemos retornar um erro.
    if len(inp) < model - 1:
        return f"Could not form a n-gram of size {model} with given input."

    # Forma-se os N-gramas de tamanho model - 1
    in_ngram = list(ngrams(inp, model - 1))

    # Pega-se o último dos N-gramas do input, pois nosso modelo irá buscar
    # o N-grama mais provável que encaixa para ele

    # Por exemplo: se a entrada é "i want to" e o nosso modelo é
    # de tamanho 2, utilizaremos o N-grama ("to")
    in_ngram = in_ngram[-1]
```

```
    # Quantidade de palavras que podem encaixar na frase
    count = 0

    # Iterando sobre cada uma das tuplas (N-grama, probabilidade) do modelo
    for wp in p:
        # Seleciona-se apenas o N-grama
        p_ngram = wp[0]

        # Remove-se a última palavra do N-grama, de modo a obter um N-grama
        # de tamanho igual ao N-grama gerado para nossa entrada
        model_ngram = p_ngram[:-1]

        # Voltando ao exemplo anterior, podemos encontrar um N-grama
        # do tipo ("to", "do"). Removendo-se a última palavra,
        # teremos um N-grama ("to"), que é igual ao nosso
        # N-grama da entrada

        # Se o N-grama do modelo é igual ao N-grama da entrada
        # tivemos um match
        if model_ngram == in_ngram:
            count += 1

            # Seleciona-se apenas a última palavra do N-grama do modelo
            # e faz-se o replace do hífen artificial pela apóstrofe
            match = p_ngram[-1].replace("-", "'")
            pred.append(match)

            # Novamente utilizando o exemplo anterior, como tivemos um
            # match, o modelo assume que queriamos o N-grama
            # ("to", "do"); logo, preenche a frase
            # "i want to" com a palavra "do"

            # Se obtivemos o número exigido de palavras, quebra o loop
            if count == n:
                break

    # Se não conseguimos obter palavras suficentes
    # preenche-se a lista com N/A
    if count < n:
        pred += ["N/A"] * (n - count)

    return pred
```

5 Testando o modelo

Para testar o modelo, utilizou-se a frase "i want to" (em letras minúsculas pois os dados estão em lowercase), escolheu-se obter as 5 primeiras palavras, e escolheu-se o modelo de trigramas.

```
example = "i want to"
pred = return_prediction(example, n=5, model=3)

for word in pred:
    print(example + " " + word)
```

```
i want to be
i want to see
```

```
i want to do
i want to go
i want to make
```

Como mencionado anteriormente porém, o modelo nem sempre é preciso. O exemplo abaixo utiliza a frase "let me see your" e um modelo de quadrigramas.

```
example = "let me see your"
pred = return_prediction(example, n=5, model=4)

for word in pred:
    print(example + " " + word)
```

```
let me see your N/A
let me see your N/A
let me see your N/A
let me see your N/A
let me see your N/A
```

6 Conclusão e iterações futuras

É possível concluir que a partir de um modelo de N-gramas é possível fazer um preditor de palavras de texto eficaz e prático. Todavia, o modelo não é muito preciso, especialmente para frases grandes e para modelos de quadrigramas. Outro problema é que o modelo não considera o contexto completo da frase, e seleciona-se apenas as últimas palavras desta para gerar a predição.

Para melhorar algumas destas falhas, pode-se considerar construir N-gramas maiores para gerar modelos mais precisos. Além disso, poderia-se implementar um sistema de retroalimentação da base de dados, de modo a fornecer mais dados.

A ferramenta também poderia receber novas features. Um exemplo interessante seria o uso de um sistema de input em tempo real, de modo que o usuário tenha uma experiência mais suave e agradável.

Além disso, pode-se implementar o uso de outras linguagens no modelo. Para tal, seriam necessários ajustes no pré-processamento e limpeza dos dados, e também seria necessário obter as stopwords da nova língua.

7 Bibliografia

[1] Old newspapers - https://www.kaggle.com/alvations/old-newspapers

[2] Documentação do NLTK - https://www.nltk.org/index.html

[3] Understanding Word N-grams and N-gram Probability in Natural Language Processing - https://towardsdatascience.com/understanding-word-n-grams-and-n-gram-probability-in-natural-language-processing-9d9eef0fa058

[4] N-gram Language Models - https://towardsdatascience.com/n-gram-language-models-af6085435eeb