PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

NÚCLEO DE EDUCAÇÃO A DISTÂNCIA

Pós-graduação *Lato Sensu* em Ciência de Dados e Big Data

Adriano Diniz Castanheira

CLASSIFICAÇÃO DE NCM COM BASE NA DESCRIÇÃO DE MERCADORIAS SUJEITAS A PENA DE PERDIMENTO

São Paulo

2021

Adriano Diniz Castanheira

CLASSIFICAÇÃO DE NCM COM BASE NA DESCRIÇÃO DE MERCADORIAS SUJEITAS A PENA DE PERDIMENTO

Trabalho de Conclusão de Curso apresentado ao Curso de Especialização em Ciência de Dados e Big Data como requisito parcial à obtenção do título de especialista.

São Paulo

2021

AGRADECIMENTOS

À minha esposa, Carolina Vieira Nogueira, que me incentivou nos momentos mais difíceis e compreendeu minha ausência para a realização deste trabalho.

Ao Nélio Machado, por ter me apresentado ao mundo da Ciência de Dados de uma maneira didática e cativante.

Ao Ivan da Silva Brasílico, que participou diretamente do desenvolvimento deste projeto de pesquisa, enriquecendo meu aprendizado.

À PUC-MG, Núcleo de Educação à Distância, essencial no meu processo de formação profissional, pela dedicação, e por tudo o que aprendi ao longo do curso.

Sumário

1. Introdução

1.1. Contextualização

Nos termos do art. 689 do Regulamento Aduaneiro (RA - Decreto 6.759/2009)[1] da Secretaria Receita Federal do Brasil, quando uma mercadoria se encontra sujeita à pena de perdimento é necessário que seja lavrado um auto de infração, acompanhado de termo de guarda fiscal. Neste termo de guarda fiscal, as mercadorias devem ser descritas, identificadas, valoradas e classificadas considerando a Tarifa Externa Comum (TEC)[2], com base na Nomenclatura Comum do Mercosul (NCM).

Como membro do Mercado Comum do Sul (Mercosul), o Brasil adota a TEC para todos códigos NCM, com exceção de códigos e partes de códigos (Ex-tarifários). A TEC tem por base o Sistema Harmonizado (SH) mantido pela Organização Mundial das Aduanas (OMA). O código NCM é utilizado então para determinar uma certa mercadoria utilizando apenas um único código. Esse código é aplicado regionalmente no âmbito do Mercosul, mas utiliza 6 regras gerais do SH e 2 regras gerais complementares. Assim o código NCM de 8 dígitos é dividido em: Capítulo (dois primeiros dígitos), Posição (terceiro e quarto dígitos), Subposição (quinto e sexto dígitos), Item (sétimo dígito) e Subitem (oitavo dígito).

Exemplo: NCM 0102.21.10

01	Capítulo	Animais Vivos
01.02	Posição	Animais Vivos da espécie bovina
0102.2	Subposição	Bovinos Domésticos
0102.21	Subposição	Reprodutores de raça pura

[1] *http://www.planalto.gov.br/ccivil_03/_ato2007-2010/2009/decreto/d6759.htm*
[2] http://www.camex.gov.br/tarifa-externa-comum-tec/tarifa-externa-comum

0102.21.10	Item e Subitem	Prenhes ou com cria ao pé

Dito isto, uma vez obtido o código NCM de uma mercadoria é possível determinar os tributos envolvidos nas operações de comércio exterior. Além disso, a NCM é base para o estabelecimento de direitos de defesa comercial, valoração aduaneira, dados estatísticos de importação e exportação, identificação de mercadorias em regimes aduaneiros especiais, entre outras.

Dentre as diversas possibilidades em que são aplicadas a pena de perdimento está a da mercadoria abandonada (art. 642 do RA). Nesta situação os Recintos Alfandegários, após cumpridos os prazos definidos, comunicam para a Receita Federal do Brasil o abandono da carga, e a Receita Federal elabora uma relação das mercadorias para consignar no termo de guarda fiscal. Em outros casos, o perdimento pode ocorrer por penalidade aplicada pela Receita Federal ao se deparar com uma infração, como as previstas no Art. 689 do RA. No intuito de descrever corretamente, atribuir valor a essa relação de mercadorias, calcular eventuais tributos devidos e dano ao erário, faz-se necessária a consulta à TEC.

1.2. O problema proposto

No contexto das mercadorias sujeitas à pena de perdimento e considerando as etapas do procedimento de lavratura do Auto de Infração, temos como uma das atividades mais relevantes a identificação da mercadoria e classificação seguindo o código NCM. Terminada a identificação, é possível atribuir um valor à mercadoria bem como tomar as medidas cabíveis de destinação dessa mercadoria.

Assim, a proposta desse trabalho é resolver de forma prática a etapa de classificação obedecendo a NCM, utilizando para isso um sistema de predição com base na descrição obtida das mercadorias. Esse sistema executará um aprendizado de máquina aplicado

no texto das descrições executando um algoritmo de Processamento de Linguagem Natural (PLN ou em Inglês - Natural Language Processing NLP).

Esse sistema de predição tornará o trabalho muito mais eficaz, uma vez que diminui o tempo de consulta à tabela de NCM, bem como torna o trabalho mais eficiente, uma vez que é possível classificar diversos itens em lote, otimizando o tempo.

2. Coleta de Dados

O presente trabalho levou em consideração três datasets diferentes. Os dois primeiros foram obtidos de banco de dados internos da Secretaria da Receita Federal do Brasil de sistemas que são utilizados diretamente no trato de mercadorias abandonadas. O terceiro dataset foi obtido na internet[3].

O primeiro dataset é oriundo do Sistema Ajna, sistema que trata de visão computacional para Aduana, e que após processamento, que será explicado no Capítulo 3 deste trabalho, forneceu 6.272 linhas de descrições de mercadoria com o respectivo código NCM. A extração dos dados foi realizada em Março de 2021 e contém dados desde Março de 2020. Esse dataset será chamado de “itenstg”, fazendo referência ao Termo de Guarda, documento utilizado na instrução dos Autos de Infrações e Multas relativos a mercadorias abandonadas (Fig. 1).

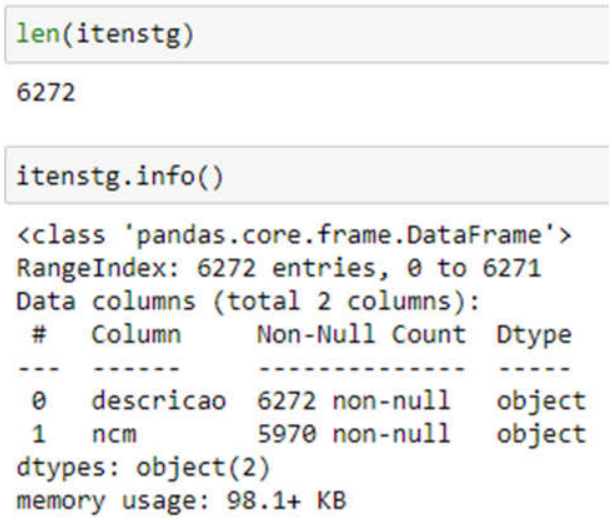

Fig. 1 - informações do dataset “itenstg”

O segundo dataset é oriundo do Sistema Secta, que trata do registro de importações e exportações em âmbito nacional, e que após processamento, também explicado no Capítulo 3 deste trabalho, forneceu aproximadamente 4 milhões linhas de descrições de

[3] http://www.camex.gov.br/tarifa-externa-comum-tec/tec-listas-em-vigor

mercadoria e o respectivo código NCM. A extração dos dados foi realizada em Março de 2021 e contém dados desde 2010. Este dataset será chamado de “secta” em alusão ao nome do sistema do qual os dados foram extraídos (Fig. 2).

```
len(secta)
```

```
4467033
```

```
secta.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4467033 entries, 0 to 4467032
Data columns (total 7 columns):
 #   Column         Dtype
---  ------         -----
 0   Data Registro  object
 1   tipo           object
 2   NCM            float64
 3   descrição      object
 4   marca          object
 5   modelo         object
 6   Observacao     object
dtypes: float64(1), object(6)
memory usage: 238.6+ MB
```

Fig. 2 - informações do dataset “secta”

O terceiro dataset é oriundo da internet e contém os dados das NCMs retirados da Tarifa Externa Comum (TEC). Após processamento, que será explicado no Capítulo 3 deste trabalho, forneceu em torno de 10 mil linhas. Foi utilizada a TEC atualizada em 31 de Março de 2021 e pré-processada utilizando o algoritmo explicado no Artigo “Utilização de tecnologia de Recuperação e Ranqueamento de Texto classificação fiscal”[4]. Este dataset será chamado de “tec” (Fig. 3).

[4] https://medium.com/@brasilico.ivan/utiliza%C3%A7%C3%A3o-das-t%C3%A9cnicas-de-recupera%C3%A7%C3%A3o-e-ranqueamento-de-texto-e-t%C3%A9cnicas-de-ia-para-aux%C3%ADlio-25d047d4fe06

```
len(tec)
```

```
10147
```

```
tec.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10147 entries, 0 to 10146
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   descricao  10147 non-null  object 
 1   ncm        10147 non-null  float64
dtypes: float64(1), object(1)
memory usage: 158.7+ KB
```

Fig. 3 - informações dataset “tec”

3. Processamento/Tratamento de Dados

Optou-se no presente trabalho por utilizar a linguagem de programação Python[5], pois fornece grande quantidade de bibliotecas de ciência de dados, e utilizou-se também o Jupyter Notebook[6] como interface de programação e apresentação dos resultados, dada sua facilidade em executar pequenas partes de um código por vez.

Os dados obtidos sofreram os seguintes processamentos:

3.1. Dataset “secta”:

Primeiramente, como esse dataset possui mais de 4 milhões de linhas, optamos por utilizar somente os registros do ano de 2021, o que totalizou 174.808 registros (Fig.4).

```
secta = secta[secta['Data Registro'] > '2021-01-01 00:00:00']

len(secta)  # tamanho do dataframe secta

174808
```

Fig.4 - limita o dataset “secta” com dados de 2021

Além disso, a fim de trabalharmos com um único campo de descrição de mercadoria, foi necessário concatenar todos os campos que traziam algum tipo de informação sobre cada mercadoria. Dessa forma, após este processamento inicial obtivemos um dataset idêntico ao do “itenstg” e do “tec” originais, sendo assim possível utilizar as mesmas técnicas de exploração e processamento de dados nos três datasets.

Contudo, antes de concatenar as colunas, foi necessário renomeá-las (Fig. 5), eliminar os campos nulos (conteúdo NaN - not a number) substituindo-os por vazio (Fig. 6) e substituir por vazio os termos genéricos “OUTRAS” da coluna “tipo” (Fig. 7) e “*[OUTROS (DEMAIS TIPOS)]” da coluna descrição (Fig. 8).

[5] https://www.python.org/
[6] https://jupyter.org/

```
# renomeia as colunas
secta = secta.rename(columns={"NCM": "ncm", "descrição": "desc", "Observacao": "observacao"})
```

Fig. 5 - renomeia colunas

```
# substitui tudo que é NaN por vazio
secta['tipo'] = secta.tipo.replace(np.nan, '', regex=True)

secta['desc'] = secta.desc.replace(np.nan, '', regex=True)

secta['marca'] = secta.marca.replace(np.nan, '', regex=True)

secta['modelo'] = secta.modelo.replace(np.nan, '', regex=True)

secta['observacao'] = secta.observacao.replace(np.nan, '', regex=True)
```

Fig. 6 - elimina valores faltantes

```
# quantidade de termos 'OUTRAS' na coluna 'tipo'
len(secta[secta['tipo'] == 'OUTRAS'])
```

```
41721
```

```
# substitui 'OUTRAS' por vazio e verifica quantidade
secta['tipo'] = secta.tipo.replace('OUTRAS', '')
len(secta[secta['tipo'] == 'OUTRAS'])
```

```
0
```

Fig. 7 - apaga informações da coluna

```
# quantidade de termos '*[OUTROS (DEMAIS TIPOS)]' na coluna 'desc'
len(secta[secta['desc'] == '*[OUTROS (DEMAIS TIPOS)]'])
```

```
4184
```

```
# substitui '*[OUTROS (DEMAIS TIPOS)]' por vazio e verifica quantidade
secta['desc'] = secta.desc.replace('*[OUTROS (DEMAIS TIPOS)]', '')
len(secta[secta['desc'] == '*[OUTROS (DEMAIS TIPOS)]'])
```

```
0
```

Fig. 8 - apaga informações das colunas

Os campos do dataset Secta que contém informações sobre a descrição das mercadorias são as colunas “tipo”, “desc”, “marca”, “modelo” e “observacao”. Dessa

forma, criou-se uma coluna nova chamada “descricao” (Fig. 9) e posteriormente sobrescrevemos o dataframe Secta criando um novo dataframe apenas com a coluna “descricao” e “ncm” (Fig. 10). Idêntico ao datasets “itenstg” e “tec”.

```
# cria nova coluna chamada 'descricao' cujo conteúdo é a concatenação
# das colunas 'desc', 'marca', 'modelo' e 'observacao'
secta['descricao'] = secta['tipo'] + ' ' + secta[
    'desc'].astype(str) + ' ' + secta['marca'].astype(str) +' '  + secta[
    'modelo'].astype(str) + ' '  + secta['observacao'].astype(str)
```

Fig. 9 - cria coluna “descricao” concatenando as colunas indicadas

```
# cria novo dataframe com as colunas 'descricao' e 'ncm'
secta = secta[['descricao','ncm']]
```

Fig. 10 - recria o dataframe “secta”

A partir deste momento, as três bases de dados contém os mesmos tipos de colunas (“descricao” e “ncm”) e dessa forma, foram processadas da mesma maneira. Ou seja, todos os tratamentos de dados explicados na sequência foram aplicados às três bases de dados. Cabe observar também que as bases foram tratadas separadamente a fim de explorar suas particularidades e depois concatenadas.

3.2 Coluna “ncm”:

O primeiro tratamento efetuado foi transformar o valor de NCM para Float uma vez que essa coluna no dataframe “itenstg” é um object (Fig. 1). Esse procedimento foi necessário, pois o dado que não é passível de ser convertido para Float, um texto por exemplo, acaba sendo convertido para NaN e excluído posteriormente (Fig. 11) como dados faltantes (no Inglês, missing values).

```
# tenta transformar em float, se não conseguir marca como NaN
for i, value in enumerate(itenstg.ncm):
    try:
        float(value)
    except Exception:
        itenstg.ncm[i] = float('NaN')
```

```
# quantidade de itens faltantes na coluna ncm
sum(itenstg.ncm.isna())
```

```
303
```

```
# apaga todas as linhas do dataframe onde
# o valor da coluna ncm é valor faltante
itenstg = itenstg.dropna(subset=['ncm'])
```

```
# tamanho do dataframe após remoção
# dos valores faltantes
len(itenstg)
```

```
5969
```

Fig. 11 - trata missing values coluna “ncm”

O passo seguinte é transformar o valor do campo “ncm” em string (texto) e salvar numa nova coluna “ncm_str”. Mas para isso converte-se primeiro em número inteiro para caso de existir algum número após o ponto flutuante do Float e converte-se esse inteiro para String. Após a conversão, preenche-se com zeros à esquerda para que o campo tenha oito dígitos (Fig. 12). Se houver algum capítulo que seja “00”, não existente na TEC, é apagado da base de dados (Fig. 13).

```
itenstg['ncm_str'] = itenstg.ncm.astype(float).astype(int).astype(str)
itenstg['ncm_str'] = itenstg.ncm_str.str.zfill(8)  # preenche zeros a esquerda de modo a termos 8 caracteres no final
```

Fig. 12 - converte para string e preenche os zeros à esquerda

```
# apaga se tiver capitulo '00' - no DF original havia uma linha de teste com esse valor
itenstg = itenstg.drop(itenstg[itenstg['capitulo'] == '00'].index)
```

Fig. 13 - apaga linhas cujo capítulo sejam “00”

Uma vez tendo o valor da ncm em string, faz-se o fatiamento (parser) dessa string para obter as colunas “capítulo”, “posicao”, “subposicao”, “item”, “subitem”. (Fig. 14).

```
# os dois primeiros dígitos da NCM
itenstg['capitulo'] = itenstg.ncm_str.str[:2]

# terceiro e quarto dígitos da NCM
itenstg['posicao'] = itenstg.ncm_str.str[2:4]

# quinto e sexto dígito da NCM
itenstg['subposicao'] = itenstg.ncm_str.str[4:6]

# sétimo dígito da NCM
itenstg['item'] = itenstg.ncm_str.str[6]

# oitavo dígito da NCM
itenstg['subitem'] = itenstg.ncm_str.str[7]
```

Fig. 14 - criação de novas colunas

Nesse ponto, temos as colunas (“capitulo”, “posicao”, “subposicao”, “item”, “subitem”) alvo do algoritmo de classificação. No Capítulo 4 deste Trabalho trataremos da análise e exploração desses dados criados.

3.3 Funções auxiliares para tratamento de texto da coluna “descricao”:

Optamos por criar funções auxiliares no tratamento do texto conteúdo da coluna descrição de modo a deixar o código mais organizado.

3.3.1 Função auxiliar “remove accents”:

A primeira função auxiliar, chamada “remove_accents”, recebe uma string e realiza a normalização Unicode. Essa normalização é a transformação da string em uma forma normal onde os caracteres têm a mesma representação binária. A normalização utilizada foi a NFDK (Normalization Form Compability Decomposition)[7] que transforma os caracteres realizando a decomposição e depois verifica a compatibilidade, por exemplo,

[7] https://en.wikipedia.org/wiki/Unicode_equivalence

a letra “á” é decomposta em “a” + “´” (acento agudo). Após essa etapa, transformamos o resultado da normalização em bytes formato ASCII, dessa forma eliminamos todos os sinais ortográficos de cada caracter (Fig. 15).

```
def remove_accents(input_str):
    nfkd_form = unicodedata.normalize('NFKD', input_str)
    only_ascii = nfkd_form.encode('ASCII', 'ignore')
    return only_ascii
```

Fig. 15 - função auxiliar “remove_accents”

3.3.2 Função auxiliar “stem_word”:

A segunda função auxiliar, chamada “stem_word”, é utilizada para retirar os prefixos e sufixos das palavras, obtendo apenas o radical, essa técnica é chamada de “stemming” em Inglês. Foi utilizada a biblioteca “NLTK”[8] que traz a classe RSLPStemmer e que por sua vez contém os radicais na língua portuguesa. A função realiza um teste para ver se a palavra da qual quer extrair o radical é diferente de uma representação de byte vazio (b’’), decodifica essa palavra no padrão ‘utf-8’, retira os prefixos e sufixos e retorna a palavra codificada em bytes (Fig. 16).

```
st = RSLPStemmer()
def stem_word(word):
    if word != b'':
        word = st.stem(word.decode("utf-8"))
        word = str.encode(word)
    return word
```

Fig. 16 - função auxiliar “stem_word”

3.3.3 Função auxiliar “cpf_ou_cnpj”:

A terceira função auxiliar, chamada “cpf_ou_cnpj”, utiliza a biblioteca de expressões regulares para identificar se a palavra que está sendo processada se parece com CPF ou CNPJ. A finalidade dessa função é excluir da análise quaisquer informações que possam identificar algum contribuinte ensejando a quebra do sigilo fiscal (Fig. 17).

[8] http://www.nltk.org/howto/portuguese_en.html

```
def cpf_ou_cnpj(string):
    if re.search(
         "^([0-9]{2}[\.]?[0-9]{3}[\.]?[0-9]{3}[\/]?[0-9]{4}[-]?[0-9]{2})|([0-9]{3}[\.]?[0-9]{3}[\.]?[0-9]{3}[-]?[0-9]{2})$",
         word):
        return True
    return False
```

Fig. 17 - função auxiliar “cpf_ou_cnpj”

3.3.4 Stopwords:

Antes de apresentar a última função auxiliar é importante salientar o tratamento que foi dado às stopwords, palavras que devem ser eliminadas da base de dados pois não acrescentam valor na hora da execução dos algoritmos. Foi utilizada as stopwords da biblioteca NLTK acrescentada de algumas outras palavras de nosso interesse como por exemplo “marca”, “ref”, “referência”, “imitação”, “modelo” e das palavras anteriores ou posteriores a essas stopwords, além de outras palavras específicas. Por exemplo, considerando a descrição “telefone celular marca Samsung 64 gb China” são irrelevantes para efetuar a classificação de NCM as palavras “marca” e a palavra posterior “Samsung”, bem como a palavra “gb” como a antecessora “64”. Além disso, a palavra específica “China” também é irrelevante para classificação. Dessa forma, criamos uma lista das stopwords irrelevantes para o algoritmo, acrescentamos nessa lista as palavras anteriores (Fig 18), as palavras posteriores concatenando esses dois conjuntos (Fig.19) e acrescentando as palavras específicas, criamos outro conjunto de outras palavras irrelevantes e concatenamos ao anterior (Fig. 20) e por fim, adicionamos esse conjunto stopwords criadas à lista de stopwords da biblioteca NLTK (Fig. 21).

```
stop_words_anterior = set(['g', 'gramas', 'kg', 'kilos', 'kilograma',
                           'kilogramas', 'quilo','quilos', 'ml', 'l',
                           'litro', 'litros', 'mm', 'cm', 'm', 'giga',
                           'gigabyte', 'gigabytes', 'gb', 'tera', 'terabyte',
                           'terabytes', 'tb'])
aux = list(stop_words_anterior)
```

```
for linha in secta['descricao']:
    linha = remove_accents(linha).decode()
    linha = linha.lower()
    for word in aux:
        if word in linha.split(): # garante que a stopword existe na linha
            posi = re.search(r'\b(' + word + r')\b', linha).start()
            if posi:
                try:
                    stop_word = linha[:posi].split()[-1]  # pega a palavra anterior à palavra procurada
                    if not stop_word in excecao:
                        stop_words_anterior.add(stop_word)
                except IndexError:
                    pass
```

Fig. 18 - trata stopwords anteriores

```
stop_words_posterior = set(['marca', 'ref', 'referencia', 'imitacao', 'modelo', 'no',
                            'n', 'volume', 'tamanho', 'origem'])
aux = list(stop_words_posterior)
excecao = ['gabinete', 'bulbo', 'eletrico', 'inflavel', 'componentes',
           'diversas', 'diversos', 'digital', 'estatuetas', 'microfibra',
          'transparentes', 'headphone', 'bracelete', 'feminino']
```

```
for linha in secta['descricao']:
    linha = remove_accents(linha).decode()
    linha = linha.lower()
    for word in aux:
        if word in linha.split():  # garante que a stopword existe na linha
            posi = re.search(r'\b(' + word + r')\b', linha).start()
            if posi:
                try:
                    stop_word = linha[posi:].split()[1]  # pega a palavra posterior à palavra procurada
                    if not stop_word in excecao:
                        stop_words_posterior.add(stop_word)
                except IndexError:
                    pass
```

```
# concatena os dois conjuntos de palavras
stop_words_itens = stop_words_posterior.union(stop_words_anterior)
```

Fig. 19 - trata stopwords posteriores e concatena conjunto de stopwords

```
# conjunto de stopwords
stop_words_outras = set(['2020', '2019', 'parte', 'kit', 'c3', 's', '', '5', '35', '65', '95', '10',
                         'brazil', '2020', '2019', 'parte', 'kit', 'xiaomi', 'huawei', '', '4gb',
                          '32gb', '64gb', '128gb', '100ml', '750ml', 'no', '9', '8', '100', 'c'])

# concatena conjuntos de palavras
stop_words_itens = stop_words_itens.union(stop_words_outras)

# tamanho do conjunto que será adicionado à lista de stopwords
# da biblioteca NLTK
len(stop_words_itens)
```

```
1488
```

Fig. 20 - acrescenta outras stopwords e concatena conjunto de stopwords

```
# recupera as stopwords da biblioteca NLTK
stopwords = nltk.corpus.stopwords.words('portuguese')

# adiciona o conjunto à lista de stopwords da biblioteca
[stopwords.append(_) for _ in stop_words_itens]

# apaga repetidas
stopwords = list(set(stopwords))

# quantidade de stopwords final
print(len(stopwords))
```

```
1689
```

Fig. 21 - adiciona as stopwords criadas às da biblioteca NLTK

3.3.5 Função auxiliar “cria_coluna_descricao”:

A última função auxiliar, chamada de “cria_coluna_descricao”, é responsável por receber um dataframe, o nome da coluna com os dados de entrada, o nome da coluna com os dados de saída, lista de stopwords e uma variável booleana para realizar ou não o stemming.

Essa função, para a devida comparação e processamento das palavras, transforma todas as letras para minúsculas e também retira todos sinais de pontuação utilizando a biblioteca de expressões regulares e retira espaços em branco utilizando método .strip() do objeto String do Python. O objetivo dessa etapa é melhorar o processamento, pois esses caracteres não fazem diferença para o algoritmo (Fig. 22).

```
lista_linha = re.split('\W+', linha[col_index].strip())
```

Fig. 22 - retira espaços e sinais de pontuação

Por fim, outra característica dessa função “cria_coluna_descricao” é retornar a quantidade de palavras por linha e a quantidade de palavras no total. Essas informações serão úteis na hora de analisar e explorar o dataset. A função completa ficou dessa forma (Fig. 23):

```
def cria_coluna_descricao(dataframe, col_origem, col_destino, stop_words, stemming=False):
    num_words = Counter()
    word_count = Counter()
    times = 0
    novas_linhas = []
    col_index = dataframe.columns.get_loc(col_origem) + 1

    for linha in dataframe.itertuples():  # para cada linha da coluna de origem
        lista_linha = re.split('\W+', linha[col_index].strip())  # exclui sinais de pontuação, inclusive espaços
        num_words[len(lista_linha)]+=1  # atualiza contador de quantidade de palavras
        nova_linha = []
        for word in lista_linha:
            word = word.lower()
            if word not in stop_words:  # verifica se não está nos stopword
                # verifica se não é CPF ou CNPJ
                if not cpf_ou_cnpj(word):
                    word = remove_accents(word)  # remove acentuação
                    if stemming:
                        word = stem_word(word)  # retorna somente o radical da palavra
                    word_count[word]+=1  # atualiza contador de palavras
                    nova_linha.append(word.decode().strip())
        if len(nova_linha) >= 0:
            novas_linhas.append(' '.join(nova_linha))

    dataframe[col_destino] = novas_linhas  # cria nova coluna do dataframe com as palavras limpas
    return num_words, word_count
```

Fig. 23 - função “cria_coluna_descricao”

3.3.6 Cria colunas que servirão de entrada para o modelo:

Dessa forma, após criar as funções auxiliares e a lista de stopwords, foi possível tratar a coluna de entrada “descricao” e transformá-la em três novas colunas que serão explicadas a seguir e que servirão de dados de entrada para os modelos de machine learning.

A primeira coluna criada é a ‘descricao_limpa’ que contém o mesmo conteúdo da coluna ‘descricao’ retirados os acentos e os sinais de pontuação. Contudo, ainda possui as

stopwords e não foram retirados os prefixos e sufixos (Fig. 24). Além disso, a função retorna o total de palavras por linha e o total de palavras na coluna resultado.

```
# cria a coluna descricao_limpa sem informar nenhuma stopword e sem fazer o stemming
num_words, word_count = cria_coluna_descricao(itenstg, 'descricao', 'descricao_limpa', "", False)
```

Fig. 24 - cria coluna “descricao_limpa”

A segunda coluna criada é a ‘descricao_limpa_sem_stopwords’, que contém o mesmo conteúdo da coluna ‘descricao’ retirados os acentos, os sinais de pontuação e as stopwords (Fig. 25).

```
# cria a coluna descricao_limpa_sem_stopwords informando as stopwords, mas sem fazer o stemming
num_words, word_count = cria_coluna_descricao(itenstg, 'descricao', 'descricao_limpa_sem_stopwords', stopwords, False)
```

Fig. 25 - cria coluna “descricao_limpa_sem_stopwords”

A terceira e última coluna criada é a ‘descricao_limpa_sem_stopwords_stemming’, que contém o mesmo conteúdo da coluna ‘descricao’ retirados os acentos, os sinais de pontuação, as stopwords e os prefixos e sufixos (Fig. 26).

```
# cria a coluna descricao_limpa_sem_stopwords_stemming informando as stopwords e fazendo o stemming
num_words, word_count = cria_coluna_descricao(itenstg, 'descricao', 'descricao_limpa_sem_stopwords_stemming', stopwords, True)
```

Fig. 26 - cria coluna “descricao_limpa_sem_stopwords_stemming”

3.4 Coluna “descricao”:

Foram criadas, em cada um dos dataframes (“itenstg”, “secta” e “tec”), três novas colunas com tratamentos diferentes para o texto da coluna descrição e em seguida o resultado foi salvo em arquivos no formato “.parquet”. Esses arquivos foram carregados em um notebook específico no qual foi implementada a consolidação das bases de dados, de modo que ao final desse processamento obtivemos três novos dataframes, cada um com uma configuração de descrição limpa, stopwords e stemming.

O primeiro procedimento para consolidar os dataframes é a leitura dos arquivos parquet (Fig. 27). Em seguida, para facilitar o processamento do modelo, optamos por limitar os dados provenientes do “secta” em 10 mil linhas aleatórias, porém usando uma semente igual a “1” para que os resultados sejam sempre os mesmos (Fig. 28). Por fim, efetuamos a concatenação dos dataframes que resultou no dataframe “itens” com 26.115 linhas (Fig 29).

```
# Carrega os dataframes
df_itens = pd.read_parquet('1_itenstg_desc_limpa.parquet')
df_tec = pd.read_parquet('2_tec_desc_limpa.parquet')
df_secta = pd.read_parquet('3_secta_desc_limpa.parquet')
```

```
# Carrega os dataframes
df_itens = pd.read_parquet('1_itenstg_desc_limpa_sem_stopwords.parquet')
df_tec = pd.read_parquet('2_tec_desc_limpa_sem_stopwords.parquet')
df_secta = pd.read_parquet('3_secta_desc_limpa_sem_stopwords.parquet')
```

```
# Carrega os dataframes
df_itens = pd.read_parquet('1_itenstg_desc_limpa_sem_stopwords_stemming.parquet')
df_tec = pd.read_parquet('2_tec_desc_limpa_sem_stopwords_stemming.parquet')
df_secta = pd.read_parquet('3_secta_desc_limpa_sem_stopwords_stemming.parquet')
```

Fig. 27 - lê os arquivos .parquet de cada dataframe criados anteriormente

```
# recria o df_secta com 10 mil linhas aleatórias - usando semente = 1
df_secta = df_secta.sample(n=10_000, random_state=1)
```

Fig. 28 - limita o dataframe “secta” em 10 mil linhas, usando semente = 1

```
# concatena os dataframes
itens = pd.concat([df_itens, df_tec, df_secta])
```

```
# tamanho do dataframe 'itens'
len(itens)
```

```
26115
```

Fig. 29 - Concatena os dataframes e verifica o tamanho final

Dessa forma, temos um dataframe “itens” só com descrições limpas, outro dataframe “itens” com descrição limpa eliminando as stopwords e um terceiro dataframe “itens” que é igual ao segundo, mas aplicando-se a técnica de stemming.

Deste ponto em diante já possuímos então os dados de entrada (input) para os modelos de aprendizado de máquina devidamente separados e que servirão de base para criação da cesta de palavras.

4. Análise e Exploração dos Dados

O presente trabalho explorou e analisou os dados dos datasets antes da consolidação explicada na Fig. 29, ou seja, foram analisados individualmente os datasets “itenstg”, “secta” e “tec”. Optamos por essa metodologia pois cada um destes datasets possuem características específicas que acabam por necessitar de análises especiais antes da concatenação. Posteriormente, efetuamos a exploração dos datasets consolidados e que serviram de entrada para os modelos de aprendizado de máquina.

Primeiramente, analisamos as colunas derivadas de NCM, pois essas são as colunas alvos do modelo (capítulo, posição, subposição, item e subitem), e em um segundo momento analisamos a coluna descrição.

4.1 Coluna “ncm”:

Para as colunas da NCM criamos um dicionário com os somatórios dos valores acumulados (Fig. 30), ordenamos em ordem decrescente (Fig. 31) e montamos um gráfico de barras com o quantitativo dos dez maiores valores (Fig. 32).

```
# cira dicionário com somatório total de itens na coluna
capitulos = {}
for value in itenstg.capitulo:
    if capitulos.get(value):
        capitulos[value] += 1
    else:
        capitulos[value] = 1
```

Fig. 30 - Cria dicionário com valores acumulados

```
# ordena dicionário em ordem decrescente de quantidade
capitulos = dict(sorted(capitulos.items(), key=lambda item: item[1],reverse=True))
capitulos
```

Fig. 31 - ordena em ordem decrescente

```
# Cria gráfico de barras
values = tec['capitulo'].value_counts()  # conta quantidade de valores da coluna capítulo
threshold = 300  # define limite inferior para exibição no gráfico (exibir 10 primeiros )
mask = values > threshold
values = values.loc[mask] # pega os valores que devem ser exibidos

# informações do gráfico
ax = values.plot(kind='bar', figsize=(14,8), title="Distribuição dos capitulos")
ax.set_xlabel("Capitulos")
ax.set_ylabel("Quantidade")
```

```
Text(0, 0.5, 'Quantidade')
```

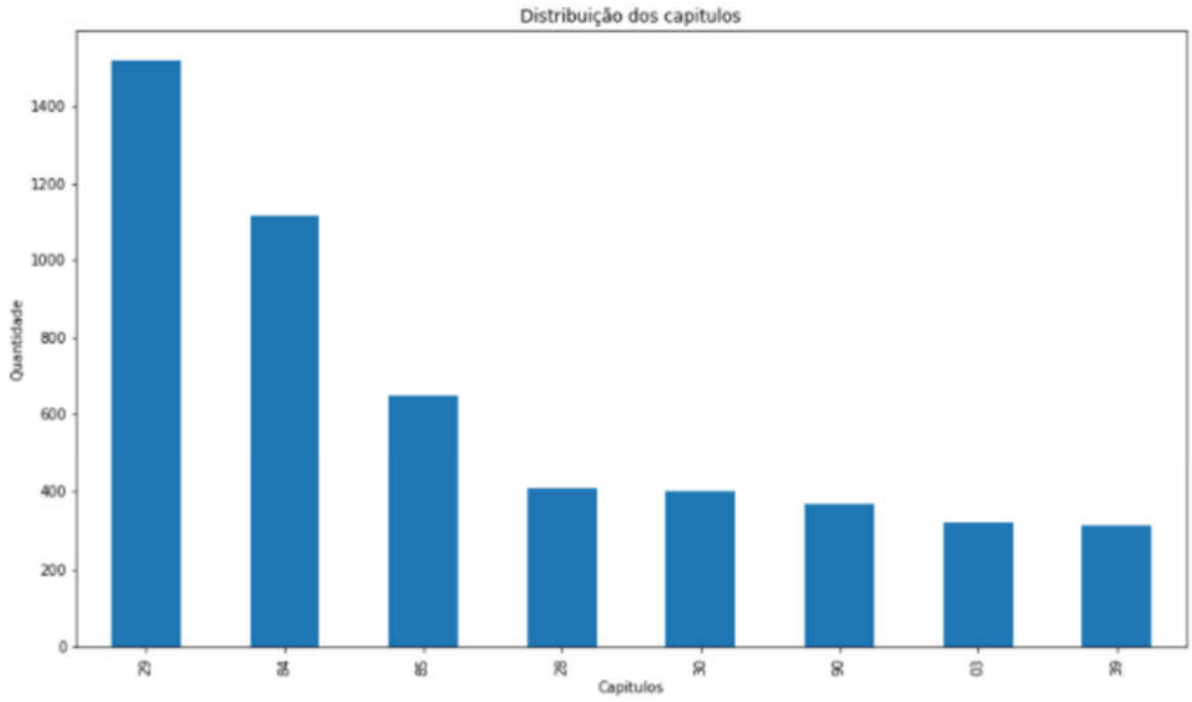

Fig .32 - código e gráfico de barras

Essa mesma análise de distribuição de frequências acumuladas foi efetuada para os campos posição, subposição, item e subitem da NCM e para os três dataframes. O resultado das distribuições de frequência acumulada considerando os dez itens mais relevantes pode ser observado na seguinte tabela:

	“itenstg”	“secta”	“tec”
capítulo	61, 84, 62, 95, 85, 42, 39, 90, 48, 33	85, 84, 33, 22, 95, 24, 90, 87, 61, 42	29, 84, 85, 28, 30, 90, 03, 39, 38, 72

posição	82, 14, 04, 06, 03, 02, 26, 10, 05, 17	17, 04, 03, 71, 18, 02, 23, 28, 07, 08	04, 02, 03, 01, 07, 06, 05, 08, 09, 10
subposição	30, 10, 50, 00, 20, 40, 90, 63, 43, 92	12, 00, 90, 62, 30, 21, 41, 10, 70, 20	90, 10, 20, 00, 30, 19, 99, 29, 11, 39
item	0, 9, 1, 2, 3, 4, 5, 7	0, 9, 1, 3, 4, 2, 7, 5, 8, 6	0, 1, 9, 2, 3, 4, 5, 6, 8, 7
subitem	0, 9, 2, 7, 1, 3, 5, 4, 6, 8	0, 9, 1, 2, 7, 3, 5, 4, 8, 6	0, 1, 9, 2, 3, 4, 5, 6, 7, 8

Cabe observar que o dataset “tec” é a fonte original de dados logo a distribuição da frequência acumulada representa a melhor distribuição possível, ou seja, é o parâmetro de referência. Os capítulos mais comuns entre as três bases foram: 90, 84 e 85. As posições mais comuns foram 02, 03 e 04; as subposições foram 00, 10, 20, 90 e 30. Os itens mais comuns são 0, 9, 1, 2 e 3 e por fim, os subitens mais comuns 0, 9, 1, 2.

As mesmas análises foram feita para o dataframe consolidado, e chegamos no seguinte resultado (Fig. 33):

```
values = itens['capitulo'].value_counts()
threshold = 450
mask = values > threshold
values = values.loc[mask]

ax = values.plot(kind='bar', figsize=(14,8), title="Distribuição dos capitulos")
ax.set_xlabel("Capitulos")
ax.set_ylabel("Quantidade")
```

```
Text(0, 0.5, 'Quantidade')
```

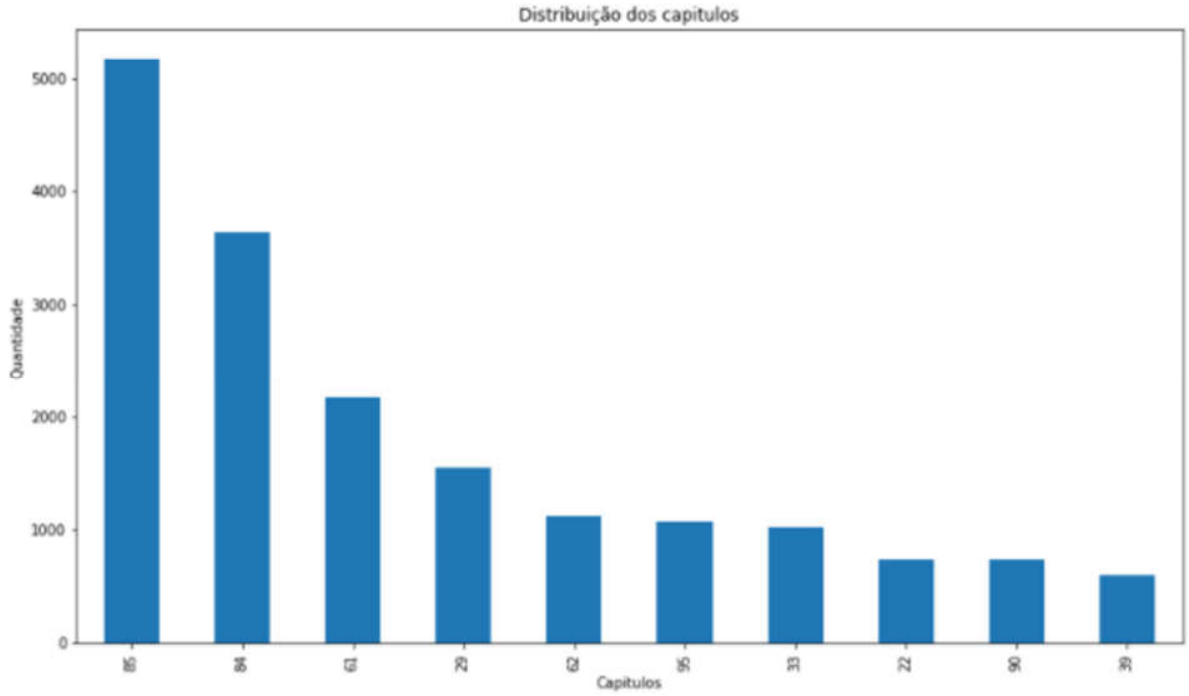

Fig. 33 - capítulos do dataframe consolidado

Efetuamos também um resumo da distribuição de frequências acumuladas, para os 10 registros mais frequentes, em cada dataframe, incluindo o dataframe consolidado, informando o capítulo, o valor total acumulado e o valor percentual (Fig. 34).

```
df_comparacao_capitulo = pd.DataFrame()
df_comparacao_capitulo['itenstg capitulo'] = df_itens['capitulo'].value_counts()[:10].index
df_comparacao_capitulo['itenstg qtidd'] = df_itens['capitulo'].value_counts()[:10].values
df_comparacao_capitulo['itenstg %'] = ((df_itens['capitulo'].value_counts()[:10].values/len(df_itens))*100).round(2)

df_comparacao_capitulo['secta capitulo'] = df_secta['capitulo'].value_counts()[:10].index
df_comparacao_capitulo['secta qtidd'] = df_secta['capitulo'].value_counts()[:10].values
df_comparacao_capitulo['secta % '] = ((df_secta['capitulo'].value_counts()[:10].values/len(df_secta))*100).round(2)

df_comparacao_capitulo['tec capitulo'] = df_tec['capitulo'].value_counts()[:10].index
df_comparacao_capitulo['tec qtidd'] = df_tec['capitulo'].value_counts()[:10].values
df_comparacao_capitulo['tec %'] = ((df_tec['capitulo'].value_counts()[:10].values/len(df_tec))*100).round(2)

df_comparacao_capitulo['itens capitulo'] = itens['capitulo'].value_counts()[:10].index
df_comparacao_capitulo['itens qtidd'] = itens['capitulo'].value_counts()[:10].values
df_comparacao_capitulo['itens %'] = ((itens['capitulo'].value_counts()[:10].values/len(itens))*100).round(2)

df_comparacao_capitulo
```

	itenstg capitulo	itenstg qtidd	itenstg %	secta capitulo	secta qtidd	secta %	tec capitulo	tec qtidd	tec %	itens capitulo	itens qtidd	itens %
0	61	1803	30.21	85	4213	42.13	29	1519	14.97	85	5183	19.85
1	84	1317	22.07	84	1212	12.12	84	1116	11.00	84	3645	13.96
2	62	911	15.26	33	910	9.10	85	649	6.40	61	2176	8.33
3	95	511	8.56	22	682	6.82	28	409	4.03	29	1549	5.93
4	85	321	5.38	95	517	5.17	30	401	3.95	62	1133	4.34
5	42	197	3.30	24	290	2.90	90	387	3.82	95	1077	4.12
6	39	179	3.00	90	244	2.44	03	320	3.15	33	1021	3.91
7	90	126	2.11	61	241	2.41	39	313	3.08	22	742	2.84
8	48	76	1.27	87	197	1.97	38	259	2.55	90	737	2.82
9	33	58	0.97	42	137	1.37	72	212	2.09	39	596	2.28

Fig. 34 - Resumo da distribuição de capítulos em cada dataframe

Conclusão, considerando o dataframe TEC como gabarito e o dataframe consolidado, temos que o capítulo 29 da TEC é o que tem mais registros, correspondendo a aproximadamente 15%. Esse valor percentual cai para aproximadamente 6% no dataframe consolidado, sendo que os capítulos mais frequentes do consolidado são 85 e 84, responsáveis por aproximadamente 34% dos registros, o que diferentemente no TEC, correspondem a aproximadamente 17%. Além disso, no dataset consolidado os capítulos 85, 84, 61, 29 e 62 representam mais da metade de todos os registros.

Essa mesma análise foi efetuada para as posições, subposições, item e subitem (Fig. 35). As cinco posições mais significativas (04, 03, 17, 02, 14) do dataframe consolidado estão coerentes com as posições dos dataframes de origem. No que se refere a subposição, a mais frequente (30) no dataframe resultante é mais relevante na origem “itenstg”, enquanto as outras mais frequentes no dataframe final são coerentes com todos dataframes iniciais. Da mesma forma, os campos “item” e “subitem” do dataframe final são coerente com os dados originais.

	itenstg posicao	itenstg qtidd	itenstg %	secta posicao	secta qtidd	secta %	tec posicao	tec qtidd	tec %	itens posicao	itens qtidd	itens %
0	82	1238	20.74	17	1997	19.97	04	771	7.60	04	2631	10.07
1	14	1090	18.26	04	1192	11.92	02	758	7.47	03	2393	9.16
2	04	668	11.19	03	1054	10.54	03	730	7.19	17	2294	8.78
3	06	658	11.03	71	835	8.35	01	506	4.99	02	1730	6.62
4	03	609	10.20	18	592	5.92	07	481	4.74	14	1518	5.81
5	02	414	6.94	02	558	5.58	06	465	4.58	06	1262	4.83
6	26	161	2.70	23	395	3.95	05	462	4.55	82	1258	4.82
7	10	141	2.36	28	357	3.57	08	436	4.30	07	905	3.47
8	05	117	1.96	07	324	3.24	09	343	3.38	71	879	3.37
9	17	105	1.76	08	244	2.44	10	297	2.93	18	857	3.28

	itenstg subposicao	itenstg qtidd	itenstg %	secta subposicao	secta qtidd	secta %	tec subposicao	tec qtidd	tec %	itens subposicao	itens qtidd	itens %
0	30	1378	23.09	12	981	9.81	90	1390	13.70	30	2627	10.06
1	10	1004	16.82	00	976	9.76	10	981	9.67	90	2599	9.95
2	50	557	9.33	90	926	9.26	20	735	7.24	10	2517	9.64
3	00	541	9.07	62	801	8.01	00	490	4.83	00	2007	7.68
4	20	389	6.52	30	779	7.79	30	470	4.63	20	1580	6.05
5	40	372	6.23	21	716	7.16	19	455	4.48	12	1280	4.90
6	90	283	4.74	41	584	5.84	99	436	4.30	50	1001	3.83
7	63	197	3.30	10	532	5.32	29	385	3.79	21	974	3.73
8	43	125	2.09	20	456	4.56	11	348	3.43	99	901	3.45
9	92	117	1.96	70	445	4.45	39	331	3.26	62	900	3.45

	tec item	tec qtidd	tec %	itenstg item	itenstg qtidd	itenstg %	secta item	secta qtidd	secta %	itens item	itens qtidd	itens %
0	0	3750	36.96	0	3560.0	59.65	0	3422	34.22	0	10732	41.09
1	1	2269	22.36	9	1768.0	29.62	9	2341	23.41	9	5794	22.18
2	9	1683	16.59	1	296.0	4.96	1	2063	20.63	1	4628	17.72
3	2	1168	11.51	2	278.0	4.66	3	1020	10.20	2	1646	6.30
4	3	524	5.16	3	59.0	0.99	4	551	5.51	3	1603	6.14
5	4	287	2.83	4	4.0	0.07	7	214	2.14	4	842	3.22
6	5	182	1.79	5	2.0	0.03	2	200	2.00	5	336	1.29
7	6	109	1.07	7	1.0	0.02	5	152	1.52	7	291	1.11
8	8	99	0.98	NaN	NaN	NaN	8	36	0.36	8	135	0.52
9	7	76	0.75	NaN	NaN	NaN	6	1	0.01	6	110	0.42

	itenstg subitem	itenstg qtidd	itenstg %	tec subitem	tec qtidd	tec %	secta subitem	secta qtidd	secta %	itens subitem	itens qtidd	itens %
0	0	5292	88.87	0	7250	71.45	0	6898.0	68.96	0	19438	74.43
1	9	298	4.99	1	861	8.49	9	1481.0	14.81	9	2602	9.96
2	2	209	3.50	9	821	8.09	1	1182.0	11.82	1	2109	8.08
3	7	97	1.63	2	487	4.80	2	214.0	2.14	2	910	3.48
4	1	66	1.11	3	282	2.78	7	106.0	1.06	3	343	1.31
5	3	2	0.03	4	164	1.62	3	59.0	0.59	7	258	0.99
6	5	1	0.02	5	111	1.09	5	28.0	0.28	4	191	0.73
7	8	1	0.02	6	75	0.74	4	26.0	0.26	5	140	0.54
8	4	1	0.02	7	55	0.54	8	8.0	0.08	6	76	0.29
9	6	1	0.02	8	41	0.40	NaN	NaN	NaN	8	50	0.19

Fig. 35 - Resumo das distribuições de posições, subposições, itens e subitens

4.2 Colunas que tratam da descrição das mercadorias:

Após a criação das colunas relativas às descrições, analisamos as palavras mais frequentes e criamos uma nuvem de palavras para visualizarmos o efeito da retirada das stopwords e do stemming nas palavras mais frequentes em cada um dos dataframes originais, bem como para o dataframe final. Essa nuvem de palavras levou em consideração as cinquenta palavras mais frequentes para não ficar uma nuvem muito difícil de visualizar.

Para gerar a nuvem de palavras, implementamos um código (Fig.36), sendo gerados uma nuvem com a descrição limpa com stopwords e sem fazer stemming (Fig. 37), outra nuvem com a descrição limpa sem stopwords (Fig. 38) e por fim uma nuvem com a descrição limpa, sem stopwords e realizando stemming(Fig. 39).

```
# cria um dicionário temporário 'text' com as 50 palavras mais frequentes
max_values = 50
text = {}
for (k, v) in word_count.most_common(max_values):
    text[str(k.decode())] = v
```

```
# cria a nuvem de palavras
wordcloud = WordCloud(width=1600, height=800).generate_from_frequencies(text)
```

```
# configurações de plotagem
fig, ax = plt.subplots(figsize=(20,10))
ax.imshow(wordcloud, interpolation='bilinear')
ax.set_axis_off()
plt.show()
```

Fig. 36 - código para gerar nuvem de palavras

Fig. 37 - nuvem de palavras “itenstg” só descrição limpa

Fig. 38 - nuvem de palavras “itenstg” sem as stopwords

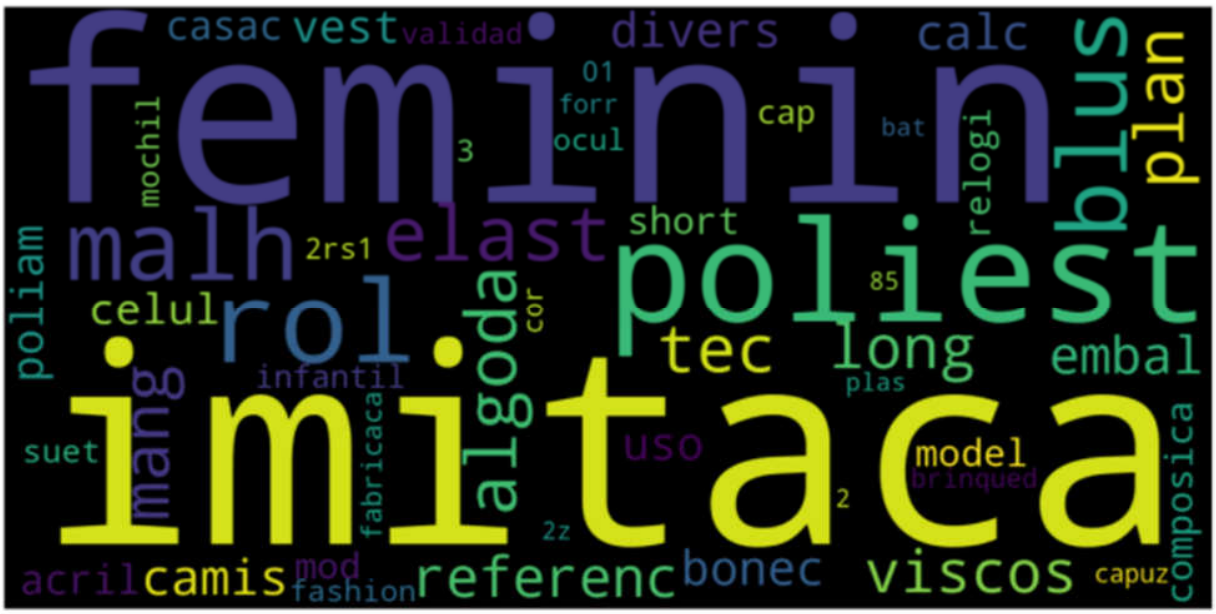

Fig. 39 - nuvem de palavras “itenstg” sem stopwords com stemming

Esse mesmo procedimento foi efetuado para os dataframes “secta”, “tec” e “itens” e estão nos respectivos notebooks. Dessa forma, pudemos ter uma ideia visual das palavras mais frequentes em cada tipo de tratamento que efetuamos nos datasets. Para facilitar a comparação entre esses diferentes tratamentos, foi criado um dataframe com essas palavras mais frequentes para resumir, conforme segue.

Dataframe “itenstg”:

	descricao_limpa	descricao_limpa_sem_stopwords	descricao_limpa_sem_stopwords_stemming
0	marca	imitacao	imitaca
1	imitacao	poliester	feminin
2	ref	rolamento	poliest
3	poliester	feminina	rol
4	de	feminino	malh
5	da	malha	blus
6	rolamento	blusa	elast
7	feminina	elastano	algoda
8	feminino	algodao	plan
9	referencia	plano	tec
10	100	tecido	mang
11	modelo	manga	long
12	malha	longa	viscos
13	blusa	viscose	referenc
14	elastano	referencia	embal
15	com	camisa	camis
16	algodao	embalagem	calc
17	plano	uso	divers
18	5	diversos	bonec
19	tecido	vestido	uso

Fig. 40 - amostra de palavras “itenstg”

Podemos observar que a retirada das stopwords permitiu o aumento da representatividade de palavras mais interessantes para o modelo de processamento de linguagem natural. Além disso, o fato de realizar o stemming permitiu um agrupamento, e consequentemente o aumento da frequência acumulada, de palavras que estavam mais dispersas no espaço amostral como por exemplo “feminino” e “feminina” que ficaram agrupadas em “feminin”.

Dataframe “secta”:

	descricao_limpa	descricao_limpa_sem_stopwords	descricao_limpa_sem_stopwords_stemming
0	de	telefones	telefon
1	eletronicos	informatica	informa
2	telefones	celular	perfum
3	xiaomi	smartphone	celul
4	informatica	bebidas	smartphon
5	redmi	telefone	beb
6	celular	perfume	vinh
7	smartphone	perfumes	vestuari
8	note	vinho	brinqued
9	china	roteador	rote
10	bebidas	receptor	recep
11	origem	vestuarios	cigarr
12	telefone	cigarros	divers
13	perfume	fone	fon
14	perfumes	pro	pro
15	s	acess	aces
16	64gb	satelite	satelit
17	128gb	acustica	acus
18	vinho	smartwatch	smartwatch
19	9	relog	relog

Fig. 41 - amostra de palavras “secta”

Da mesma forma que no dataframe “itenstg” a retirada das stopwords permitiu maior relevância para outras palavras que haviam sido ofuscadas pela quantidade enorme de stopwords e outros termos adicionados como china, xiaomi, 64gb, etc. Além disso, o fato de realizar o stemming permitiu um agrupamento, e consequentemente o aumento da frequência acumulada, de palavras que estavam mais dispersas no espaço amostral como por exemplo “telefones” e “telefone”, “perfumes” e “perfume”.

Dataframe “tec”:

	descricao_limpa	descricao_limpa_sem_stopwords	descricao_limpa_sem_stopwords_stemming
0	de	outros	outr
1	e	nao	nao
2	ou	outras	aparelh
3	outros	aparelhos	inclu
4	para	incluindo	produt
5	a	produtos	excet
6	nao	exceto	maquin
7	em	maquinas	deriv
8	outras	derivados	dci
9	aparelhos	dci	superi
10	os	superior	semelh
11	seus	semelhantes	sal
12	incluindo	contenham	contenh
13	produtos	sais	posico
14	mesmo	posicoes	part
15	por	posicao	mat
16	com	partes	fio
17	exceto	materias	acid
18	maquinas	artigos	posica
19	que	peixes	peix

Fig. 42 - amostra de palavras “tec”

Considerando que o dataframe “tec” é a fonte principal de consulta, logo ele é o local onde todas as descrições e códigos NCMs estão corretos. Tendo isso em vista, ignorando as stopwords, as palavras mais frequentes contém algumas que não são relevantes para o algoritmo como “a”, “os”, “e”, “em”, “não”, “que”. Neste dataframe apenas utilizamos as stopwords tradicionais da biblioteca NLTK e chegamos em um resultado bastante interessante, já que sobraram palavras relevantes ao modelo de aprendizado de máquina. Além disso, o fato de realizar stemming não surtiu muito efeito, mantendo a distribuição muito parecida com a base sem stopwords.

Dataframe “itens”:

	descricao_limpa	descricao_limpa_sem_stopwords	descricao_limpa_sem_stopwords_stemming
0	de	outros	outr
1	e	nao	nao
2	ou	outras	aparelh
3	outros	aparelhos	inclu
4	para	incluindo	produt
5	a	produtos	imitaca
6	em	imitacao	telefon
7	nao	exceto	maquin
8	marca	maquinas	excet
9	outras	poliester	feminin
10	aparelhos	derivados	poliest
11	com	dci	deriv
12	da	telefones	dci
13	os	superior	rol
14	seus	semelhantes	superi
15	incluindo	contenham	semelh
16	produtos	sais	sal
17	imitacao	posicoes	part
18	por	partes	contenh
19	mesmo	informatica	posico

Fig. 43 - amostra dataframe consolidado “itens”

Por fim, podemos observar que o dataframe consolidado muito se aproxima do dataframe da TEC, ou seja, temos uma base de treinamento bastante parecida com a base de dados gabarito.

5. Criação de Modelos de Machine Learning

Optamos por uma abordagem tradicional de Processamento de Linguagem Natural (PLN) na qual criamos um corpus com uma cesta de palavras (no Inglês, bag of words), fizemos o embeding, one hot encoding, TF-IDF e montamos uma rede neural utilizando a API Keras do Tensorflow[9].

Rodamos esse modelo com as três bases anteriormente definidas (descrição limpa, descrição limpa sem stopwords e descrição limpa sem stopwords stemming) e alteramos alguns parâmetros de configuração da arquitetura dos modelos para chegar num resultado de melhor acurácia. Para isso, optamos por classificar primeiramente a coluna "capítulo" até encontrarmos a melhor configuração. Em seguida, rodamos esse modelo para classificar todas as outras partes da NCM (posição, subposição, item, etc)..

5.1 - Baseline Keras:

Primeiramente criamos uma função, chamada "encoded_fields" (Fig. 44), que recebe um dataframe e as colunas que serão transformadas. Basicamente essa função analisa uma coluna do dataframe criando um valor numérico para cada categoria (label encoder) de tamanho 'n' e cria uma matriz chamada 'one hot encoder' onde cada linha é uma dessas categorias em valor binário zero ou um, tamanho 'n x n' (label binarized) (Fig. 45). Uma vez criada essa matriz binária de one hot encoder, podemos aplicar a transformação para todos os dados da coluna enviada para a função. Se a coluna tem tamanho 'm' ao final do processo teremos um encoder de tamanho 'm'x'n', onde 'm' é a quantidade de linhas nessa coluna e 'n' é a quantidade de categorias identificadas na fase de encoder.

[9] https://www.tensorflow.org/guide/keras?hl=pt-br

```python
Encoders = namedtuple('Encoders', 'encoder binarizer')

def encode_fields(df, fields: list) -> Dict[str, Encoders]:
    result = {}
    for i, field in enumerate(fields):
        lblencoder = LabelEncoder()  # cria um número para cada categoria
        lblbinarizer = LabelBinarizer()  # one hot encoder (ex: 99categorias cria matriz com 99 intens com 0 e 1)
        encoded = lblencoder.fit_transform(df[field].values)  # transforma os dados do array "field" em classes
        # dessa forma, retorna um array com as categorias na forma numérica, começando em zero
        print(f'field: {field} / encoded shape: {encoded.shape}')
        binarized = lblbinarizer.fit_transform(encoded)  # transforma os dados do array "field" numa matriz com
        # dados binários (zeros e uns) para cada categoria, então retorna matriz m x n, onde m é a quantidade
        # de linhas do array de entrada e n é a quantidade de colunas de categorias tranformadas em sua
        # forma binária
        print(f'field: {field} / binarized shape: {binarized.shape}')
        encoders = Encoders(lblencoder, lblbinarizer)
        result[field] = encoders
    return result
```

Fig. 44 - função encode_fields

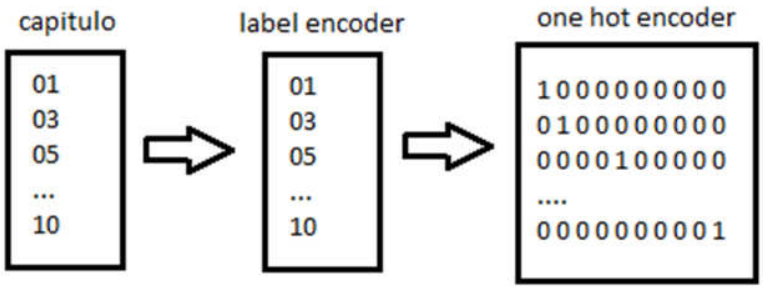

Fig. 45 - exemplo de transformação one hot encoder

Após esse procedimento, temos as colunas alvos já mapeadas para o formato binário conforme as categorias que queremos prever. Vamos trabalhar nos dados de entrada.

A entrada deste modelo é chamada de cesta de palavras (bag of words). O tratamento desta entrada consiste em pegar todo o texto e criar um corpus (conjunto de todas as palavras) de onde serão extraídos as palavras para classificação. Cada palavra é chamada de token. Em seguida, cria-se uma matriz parecida com o one hot encoder, mas ao invés de categorias, será uma linha para cada palavra. Supondo que o corpus tenha ‘n’ palavras e a entrada tenha ‘m’ linhas, ao final da chamada vetorização da cesta de palavras, teremos uma matriz tamanho ‘m x n’. No presente trabalho, a cesta de palavras, ou dicionário do modelo, contém 24.548 palavras (Fig. 46

Existem alguns procedimentos importantes nessa etapa, como eliminar palavras muito e pouco frequentes (Fig. 46). Bem como, realizar a transformação TF-IDF (do inglês term frequency–inverse document frequency) que significa frequência do termo–inverso da frequência nos documentos, ou seja, o que antes era representados por zeros e uns, agora é atribuído um peso considerando a frequência de cada palavra numa linha ponderando com o inverso da frequência desta palavra no documento todo (Fig. 47).

```
corpus = df_itens.descricao_limpa.values  # transforma todo o texto em bag of words
vectorizer = CountVectorizer(max_df=0.1, min_df=0.00001)  # elimina palavras muito ou pouco frequentes
X_counts = vectorizer.fit_transform(corpus)  # aprende o dicionário de vocabulários e gera matriz
X_counts.shape
```

```
(26117, 24548)
```

Fig. 46 - cria corpus e bag of words eliminando palavras conforme frequências

```
transformer = TfidfTransformer()  # transforma em matrix TFIDF - faz freq relativa multiplicando um
# peso nas palavras freq ou raras - ou seja, deixa de ser zero e 1.
X_tf = transformer.fit_transform(X_counts)
```

Fig. 47 - realiza a transformação TF-IDF

Para a criação do modelo foi levada em consideração que para identificarmos relações não lineares, duas camadas de neurônios são suficientes e não deixam o modelo muito complexo. Além disso, como o número máximo de categorias são as do capítulo da NCM, 99 capítulo, faz sentido trabalhar com uma densidade de neurônios com o dobro, então optamos por utilizar a densidade igual a 256. Porém, a título de mensurar o melhor modelo, variamos a densidade entre 256 e 512.

Por fim, para a criação do modelo e com o intuito de reduzir o overfiting, que é adaptação do modelo aos dados de treino tendo pouco resultado com dados diferentes, utilizamos uma camada de dropout, variando entre 20 e 40%. O dropout atua atribuindo aleatoriamente frequência igual a zero para alguns dados durante o treinamento.

Cabe observar que utilizamos o otimizador da rede neural chamado “Adam” que atualmente é utilizado para ajudar o modelo a convergir mais rápido usando a memória do computador de forma mais adequada. Usamos também como métrica a acurácia do

modelo e ativação “relu” nas camadas de entrada e escondida (hidden layer) e ativação “softmax” na camada de saída.

Dito isso, criamos uma função chamada “model1” (Fig. 48) que contém todas as características explicadas acima e que recebe como entrada as variáveis com os tamanhos de entrada e saída dos dados de treinamento (que serão utilizados para definir o tamanho da camada de entrada e saída da rede neural), bem como os valores de densidade e dropout para o modelo.

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import backend as K

def model1(input_size, output_size, optimizer='adam', dropout=0.4, dense=128):  # "adam" converge mais rápido e melhor
    # relação custo memória e tempo, é mais usado que SGD e SGD com momentum
    model = tf.keras.Sequential()
    model.add(layers.Input(input_size))
    model.add(layers.Dense(dense, activation='relu'))  # ativador do neuronio função relu
    model.add(layers.Dropout(dropout))  # a cada passada ignora % dos neurônios
    model.add(layers.Dense(dense, activation='relu'))
    model.add(layers.Dropout(dropout))
    model.add(layers.Dense(output_size, activation='softmax'))  # coleção de 0, 1 do sigmoide
    model.compile(optimizer=optimizer,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model
```

Fig. 48 - função “model1” utilizada para criar os modelos

Conforme explicado anteriormente, criamos quatro modelos para encontrar qual seria a melhor configuração da rede neural para o propósito deste trabalho (Fig. 49). Alternamos a densidade da rede entre 128 e 256 neurônios e o dropout entre 20 e 40%.

```
models = {}
dense = [128, 256]
dropout = [0.2, 0.4]
for dns in dense:
    for drpt in dropout:
        key = str(dns) + "_" + str(drpt)
        models[key] = model1(
            X_train.shape[1],
            y_train.shape[1],
            optimizer=tf.keras.optimizers.Adam(lr=0.001),
            dropout=drpt, dense=dns)
```

Fig. 49 - configurações de modelos para a rede neural

Efetuamos o treinamento destes modelos utilizando a descrição como entrada da rede e a saída foi a classificação dos capítulos da NCM. Além disso, rodamos um treinamento de 40 épocas, diminuindo a taxa de aprendizado (learning rate) a cada época e dividindo os dados de treinamentos em pedaços (chunks) de tamanho (batch) 512 registros a cada rodada (Fig. 50).

```
from collections import defaultdict
import math

epochs = 40
# batch_size = 256
batch_size = 512
rounds = X_train.shape[0] // batch_size  +  1
history = defaultdict(list)
X_val_array = X_val.toarray()
for key, model in models.items():
    print(f'\n\n modelo: {key} \n\n')
    for i in range(epochs):
        lr = 0.001 / (math.sqrt(i) + 1)
        print(f'Epoch {i} learning rate {lr}')
        K.set_value(model.optimizer.lr, lr)
        for batch_number in range(rounds):
            start = batch_number * batch_size
            X_chunk = X_train[start: start + batch_size].toarray()
            y_chunk = y_train[start: start + batch_size]
            model.train_on_batch(X_chunk, y_chunk)  # treina o modelo efetivamente
            if batch_number % 100 == 0.:
                print(f'Batch n.: {batch_number} de {rounds}')
                loss_acc = model.evaluate(X_chunk, y_chunk)
                history['train_loss'].append(loss_acc[0])
                history['train_acc'].append(loss_acc[1])
                val_loss_acc = model.evaluate(X_val_array, y_val)
                history['val_loss'].append(val_loss_acc[0])
                history['val_acc'].append(val_loss_acc[1])
                # print('loss: {:0.2f} acc: {:0.2f}'.format(val_monitor[0], val_monitor[1]))
        print('################################################')
        print(f'Final da época {i}')
        model.evaluate(X_chunk, y_chunk)
        model.evaluate(X_val.toarray(), y_val)
        print('################################################')
        del X_chunk
        del y_chunk
```

Fig. 50 - treinamento da rede neural

Por fim, analisamos a acurácia do modelo comparando os dados de treinamento com os dados de validação (Fig. 51).

```
for key, model in models.items():
    print(f'model: {key} - metrics: {model.metrics_names}')
    loss, acc = model.evaluate(X_val.toarray(), y_val)
    loss_teste, acc_teste = model.evaluate(X_train[:1000].toarray(), y_train[:1000])
    print(f'(acc_teste - acc): {(acc_teste - acc)*10_000:.2f}')
    print(f'(loss_teste - loss): {(loss - loss_teste)*100:.2f}\n')
```

Fig. 51 - análise das acurácias

Como resultado (Fig. 52) podemos observar que a melhor configuração é a rede de densidade 256 neurônios e dropout 20% (model 256_0.2), pois obteve uma acurácia e perda (loss) no treinamento de 93,57% e 0,2285, respectivamente e uma acurácia e perda na validação de 99,70% e 0,0201. O modelo treinado com densidade 256 e dropout de 40% teve uma acurácia um pouco mais elevada, contudo a perda (loss) foi maior do que o dropout de 20%.

```
for key, model in models.items():
    print(f'model: {key} - metrics: {model.metrics_names}')
    loss, acc = model.evaluate(X_val.toarray(), y_val)
    loss_teste, acc_teste = model.evaluate(X_train[:1000].toarray(), y_train[:1000])
```

```
model: 128_0.2 - metrics: ['loss', 'accuracy']
41/41 [==============================] - 0s 5ms/step - loss: 0.2569 - accuracy: 0.9311
32/32 [==============================] - 0s 5ms/step - loss: 0.0643 - accuracy: 0.9900
model: 128_0.4 - metrics: ['loss', 'accuracy']
41/41 [==============================] - 0s 4ms/step - loss: 0.3185 - accuracy: 0.9234
32/32 [==============================] - 0s 4ms/step - loss: 0.1351 - accuracy: 0.9800
model: 256_0.2 - metrics: ['loss', 'accuracy']
41/41 [==============================] - 0s 7ms/step - loss: 0.2285 - accuracy: 0.9357
32/32 [==============================] - 0s 7ms/step - loss: 0.0201 - accuracy: 0.9970
model: 256_0.4 - metrics: ['loss', 'accuracy']
41/41 [==============================] - 0s 7ms/step - loss: 0.2407 - accuracy: 0.9372
32/32 [==============================] - 0s 7ms/step - loss: 0.0394 - accuracy: 0.9950
```

Fig. 52 - comparação das configurações de redes neurais

Conforme concluímos anteriormente (Fig. 52), o melhor modelo de processamento de linguagem natural para classificar os campos da NCM tem como configuração uma rede neural de densidade de 256 neurônios e dropout de 20%. A entrada do modelo é a descrição da mercadoria e a saída optamos por dividir de acordo com cada parte da NCM. Dessa forma, rodamos um modelo com a saída sendo a classificação quanto aos capítulos, outro modelo com saída classificando quanto às posições, até chegarmos num modelo para os subitens.

Foi preciso, assim, criar encoders para cada uma das saídas, já que cada uma possui uma quantidade de categorias diferentes (Fig. 53).

```
y_cap = encoders['capitulo'].binarizer.fit_transform(y_encoded_cap)
y_pos = encoders['posicao'].binarizer.fit_transform(y_encoded_pos)
y_subpos = encoders['subposicao'].binarizer.fit_transform(y_encoded_subpos)
y_item = encoders['item'].binarizer.fit_transform(y_encoded_item)
y_subitem = encoders['subitem'].binarizer.fit_transform(y_encoded_subitem)
y_todos = [y_cap, y_pos, y_subpos, y_item, y_subitem]
```

Fig. 53 - criação de encoder para cada saída

Em seguida, criamos os dados de treino e de validação para cada uma das saídas que buscamos (Fig. 54). Cabe observar que optamos por separar 5% da base para teste, bem como utilizamos uma semente (random_state) para que os valores gerados sejam sempre os mesmos.

```
X_train_cap, X_val_cap, y_train_cap, y_val_cap = train_test_split(X_tf, y_cap, test_size=0.05, stratify=y_cap, random_state=1)

X_train_pos, X_val_pos, y_train_pos, y_val_pos = train_test_split(X_tf, y_pos, test_size=0.05, random_state=1)

X_train_subpos, X_val_subpos, y_train_subpos, y_val_subpos = train_test_split(X_tf, y_subpos, test_size=0.05, random_state=1)

X_train_item, X_val_item, y_train_item, y_val_item = train_test_split(X_tf, y_item, test_size=0.05, stratify=y_item, random_stat

X_train_subitem, X_val_subitem, y_train_subitem, y_val_subitem = train_test_split(X_tf, y_subitem, test_size=0.05, stratify=y_sul
```

Fig. 54 - separa as bases de treino e de validação

Dessa forma, os modelos que serão executados possuem as seguintes configurações de dados de entrada (X_Train) e categorias de saída (y_train) (Fig. 55):

```
model cap: Xtrain: 24548 e y_train: 97
model pos: Xtrain: 24548 e y_train: 90
model subpos: Xtrain: 24548 e y_train: 91
model item: Xtrain: 24548 e y_train: 10
model subitem: Xtrain: 24548 e y_train: 10
```

Fig. 55 - configurações dos modelos

Configuramos também o cálculo da predição do modelo testando como entrada toda a base de dados (Fig. 56)

```
preds_list = []
for key, setup in setups.items():
    teste = df_itens.descricao_limpa.values
    preds_list.append(models[key].predict(vectorizer.transform(teste)))
```

Fig. 56 - realiza predições

6. Apresentação dos Resultados

6.1 Bnaseline Keras:

Após configurar os modelos (Fig. 54 e 55) e calcular as predições (Fig. 56), rodamos o modelo para todas as entradas de dados e criamos novas colunas com os resultados (Fig.57). Reagrupamos as partes da NCM para formar uma NCM original e a NCM resultado (Fig. 58).

```
i = 0
for key, model in models.items():
    name = key + '_resul'
    print(name)
    if key == 'cap':
        df_itens[name] = encoders['capitulo'].encoder.inverse_transform(encoders['capitulo'].binarizer.inverse_transform(preds_li
    elif key == 'pos':
        df_itens[name] = encoders['posicao'].encoder.inverse_transform(encoders['posicao'].binarizer.inverse_transform(preds_list
    elif key == 'subpos':
        df_itens[name] = encoders['subposicao'].encoder.inverse_transform(encoders['subposicao'].binarizer.inverse_transform(pred
    elif key == 'item':
        df_itens[name] = encoders['item'].encoder.inverse_transform(encoders['item'].binarizer.inverse_transform(preds_list[i]))
    elif key == 'subitem':
        df_itens[name] = encoders['subitem'].encoder.inverse_transform(encoders['subitem'].binarizer.inverse_transform(preds_list
    i += 1
```

Fig. 57 - criação de novas colunas com valores preditos

```
df_itens['ncm'] = df_itens['capitulo'] + df_itens['posicao'] + df_itens['subposicao'] + df_itens['item'] + df_itens['subitem']
```

```
df_itens['ncm_resul'] = df_itens['cap_resul'] + df_itens['pos_resul'] + df_itens['subpos_resul'] + df_itens['item_resul'] + df_it
```

Fig. 58 - reagrupamento das partes da NCM original e do resultado

Após a criação da coluna de resultado criamos um dataframe chamado “df_erros” que contém somente os registros de NCM que são diferentes dos registros de NCM resultado (Fig. 59) e obtendo como resultado um dataframe completo (Fig. 60).

```
df_erros = df_itens[df_itens['ncm'] != df_itens.ncm_resul]
```

```
print(f'Tamanho do dataset: {len(df_itens)} registros')
print(f'Quantidade de erros: {len(df_erros)}, o que representa {(len(df_erros)/len(df_itens))*100:.2f}%\n')
```

```
Tamanho do dataset: 26117 registros
Quantidade de erros: 6041, o que representa 23.13%
```

Fig. 59 - criação do “df_erros” e apresentação da quantidade

```
df_erros.head()
```

	descricao_limpa	capitulo	posicao	subposicao	item	subitem	cap_resul	pos_resul	subpos_resul	item_resul	subitem_resul	ncm	ncm_resul
9	tambor de metal d c 25kg de pasta de pigmento ...	32	19	90	3	0	32	19	90	1	0	32199030	32199010
105	chaveiro imitacao da marca kipling	95	03	00	3	1	95	03	00	3	0	95030031	95030030
124	caixa para relogio imitacao da marca diesel	42	02	99	0	0	42	02	99	9	0	42029900	42029990
126	caixa para relogio imitacao da marca emporio a...	42	02	99	0	0	42	02	99	1	0	42029900	42029910
127	certificado de garantia imitacao da marca rolex	49	11	10	1	0	49	11	10	9	0	49111010	49111090

Fig. 60 - “df_erros” que é o dataframe final para análise dos resultados

Com o dataframe de erros devidamente configurado, calculamos os erros em cada uma das partes da NCM (Fig. 61) e realizamos uma análise estatística contando a quantidade de categorias erradas e quantidade de erros em cada categoria (Fig. 62) e plotamos um gráfico com as categorias com maior número de erros (Fig. 63).

```
capitulos_err = list(df_erros[df_erros['capitulo'] != df_erros.cap_resul]['cap_resul'])

posicoes_err = list(df_erros[(df_erros['capitulo'] == df_erros.cap_resul) &
                             (df_erros['posicao'] != df_erros.pos_resul)]['pos_resul'])

subposicoes_err = list(df_erros[(df_erros['capitulo'] == df_erros.cap_resul) &
                                (df_erros['posicao'] == df_erros.pos_resul) &
                                (df_erros['subposicao'] != df_erros.subpos_resul)]['subpos_resul'])

itens_err = list(df_erros[(df_erros['capitulo'] == df_erros.cap_resul) &
                          (df_erros['posicao'] == df_erros.pos_resul) &
                          (df_erros['subposicao'] == df_erros.subpos_resul) &
                          (df_erros['item'] != df_erros.item_resul)]['item_resul'])

subitens_err = list(df_erros[(df_erros['capitulo'] == df_erros.cap_resul) &
                             (df_erros['posicao'] == df_erros.pos_resul) &
                             (df_erros['subposicao'] == df_erros.subpos_resul) &
                             (df_erros['item'] == df_erros.item_resul) &
                             (df_erros['subitem'] != df_erros.subitem_resul)]['subitem_resul'])
```

```
print(len(capitulos_err), len(posicoes_err), len(subposicoes_err), len(itens_err), len(subitens_err))
```

```
145 524 2867 1553 952
```

Fig. 61 - Analisa erros em cada parte da NCM

```
total_err = {}
for capitulo in capitulos_err:
    if total_err.get(capitulo):
        total_err[capitulo] += 1
    else:
        total_err[capitulo] = 1

print(f'Total de capítulos errados: {len(total_err.keys())}')
print(f'Total de erros em capítulos: {len(capitulos_err)} erros\n')

for k, v in total_err.items():
    total_value = len(df_itens[df_itens['capitulo'] == str(k).zfill(2)])
    print(f'Capítulo com erro: {k} => {v} erros em {total_value} = {((v/total_value)*100):.2f}%')
```

```
Total de capítulos errados: 20
Total de erros em capítulos: 145 erros

Capítulo com erro: 42 => 9 erros em 357 = 2.52%
Capítulo com erro: 62 => 29 erros em 1133 = 2.56%
Capítulo com erro: 39 => 5 erros em 596 = 0.84%
Capítulo com erro: 61 => 29 erros em 2176 = 1.33%
Capítulo com erro: 24 => 1 erros em 309 = 0.32%
Capítulo com erro: 94 => 1 erros em 100 = 1.00%
Capítulo com erro: 64 => 2 erros em 136 = 1.47%
Capítulo com erro: 49 => 1 erros em 40 = 2.50%
Capítulo com erro: 11 => 1 erros em 30 = 3.33%
Capítulo com erro: 68 => 1 erros em 77 = 1.30%
Capítulo com erro: 99 => 4 erros em 4 = 100.00%
Capítulo com erro: 87 => 2 erros em 371 = 0.54%
Capítulo com erro: 71 => 1 erros em 135 = 0.74%
Capítulo com erro: 84 => 31 erros em 3645 = 0.85%
Capítulo com erro: 85 => 18 erros em 5183 = 0.35%
Capítulo com erro: 95 => 3 erros em 1077 = 0.28%
Capítulo com erro: 90 => 3 erros em 737 = 0.41%
Capítulo com erro: 56 => 1 erros em 99 = 1.01%
Capítulo com erro: 38 => 2 erros em 310 = 0.65%
Capítulo com erro: 70 => 1 erros em 112 = 0.89%
```

Fig 62. Exemplo da análise de erros em cada categoria do Capítulo

```
# Cria gráfico de barras
df_temp = pd.DataFrame()
df_temp = df_erros[df_erros['capitulo'] != df_erros.cap_resul]
values = df_temp['cap_resul'].value_counts()
threshold = 1  # define limite inferior para exibição no gráfico (exibir 10 primeiros )
mask = values > threshold
values = values.loc[mask] # pega os valores que devem ser exibidos

# informações do gráfico
ax = values.plot.bar(figsize=(14,8), title="Distribuição de erros por capitulos")
ax.set_xlabel("Capitulos")
ax.set_ylabel("Quantidade")
print(f"Quantidade de capítulos errados: {len(df_temp['cap_resul'].value_counts())}")
```

```
Quantidade de capítulos errados: 20
```

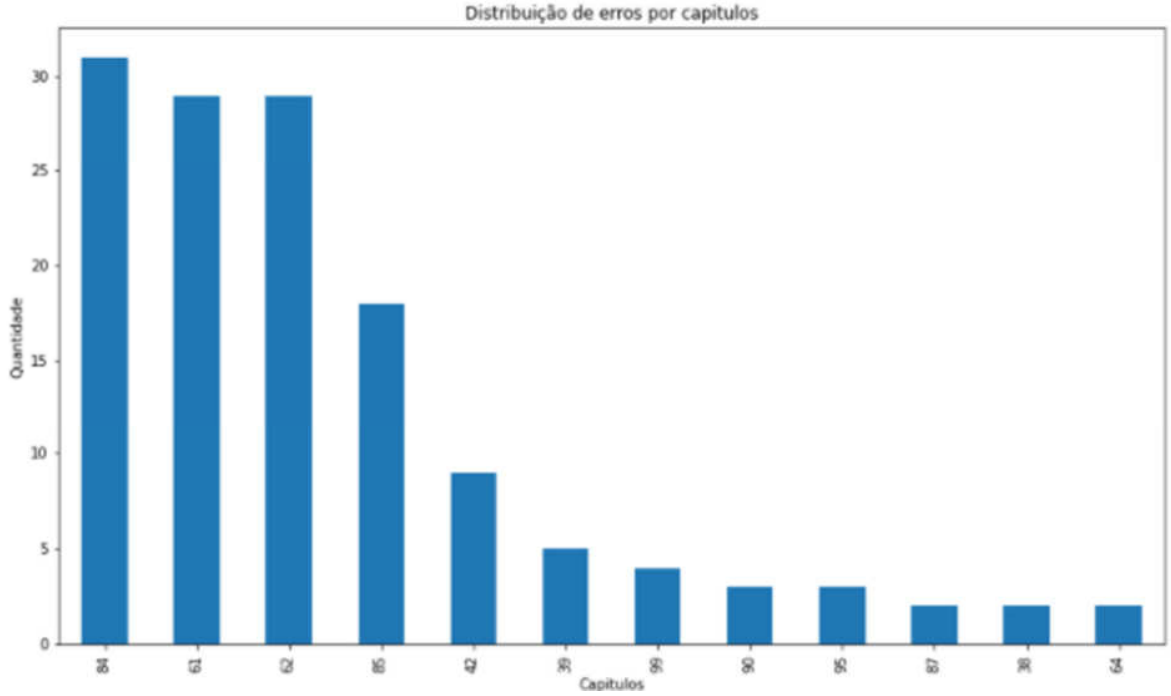

Fig. 63 - Gráfico de barra com Capítulos com maior número de erros

Consolidando todos os resultados das partes da NCM obtivemos as seguintes acurácias e perdas, resumidas na tabela abaixo (Tabela 1):

	descricao_limpa	descricao_limpa_sem_stopwords	descricao_limpa_sem_stopwords_stemming
capítulo	Acurácia: 94,70% Perda: 0,2031	Acurácia: 94,95% Perda: 0,1923	Acurácia: 94,26% Perda: 0,2063

posição	Acurácia: 91,81% Perda: 0,2964	Acurácia: 89,97% Perda: 0,3655	Acurácia: 89,05% Perda: 0,4195
subposição	Acurácia: 72,82% Perda: 1,1148	Acurácia: 68,30% Perda: 1,2575	Acurácia: 67,61% Perda: 1,2774
item	Acurácia: 78,79% Perda: 0,9768	Acurácia: 77,95% Perda: 0,9352	Acurácia: 76,57% Perda: 0,9540
subitem	Acurácia: 85,91% Perda: 0,6016	Acurácia: 85,22% Perda: 0,6356	Acurácia: 84,23% Perda: 0,6159

Tabela 1 - dados de treino

Aplicando esse mesmo modelo para todos os registros da base de dados, obtivemos os seguintes resultados (Tabela 2):

	descricao_limpa	descricao_limpa sem stopwords	Descricao_limpa sem stopwords stemming
capítulo	20 capítulos com erros 145 erros total	33 capítulos com erros 210 erros total	48 capítulos com erros 298 erros total
posição	33 posições com erros 524 erros total	43 posições com erros 468 erros total	42 posições com erros 588 erros total
subposição	47 subposições com erros 2867 erros total	46 subposições com erros 3257 erros total	49 subposições com erros 3484 erros total
item	10 itens com erros 1553 erros total	10 itens com erros 1340 erros total	10 itens com erros 1437 erros total

subitem	10 subitens com erros 952 erros total	9 subitens com erros 938 erros total	9 subitens com erros 1007 erros total
Total 26.117	Total erros: 6.041 (23,13%)	Total erros: 6.213 (23,79%)	Total erros: 6.814 (26,09%)

Tabela 2 - resultado do modelo aplicado às bases completa

6.2. Conclusões:

- Da Fig. 62, podemos notar que em relação ao Capítulo, modelo treinado com a descrição limpa, o erro por categoria foi bastante baixo. Apenas cinco capítulos, em vinte capítulos, tiveram erros maiores de 2%. Além disso, dez capítulos tiveram erros acima de 1% da amostragem, ou seja, metade dos erros por capítulo são abaixo de 1%.

- Dessa forma, supondo que o erro humano por capítulo seja na ordem de 1%, o modelo treinado teve um desempenho igual ou melhor do que um ser humano para alguns capítulos.

- Cabe observar também que os capítulos 61 e 62 possuem a peculiaridade de terem a descrição muito semelhante, apenas se diferenciando pela expressão “de malha”. Isso posto, é normal que ocorram erros de classificação entre esses dois capítulos.

- Outra informação relevante em relação à base de dados é que os erros do capítulo 99 se deve pela descrição que está bastante incompleta (Fig. 64).

```
df_itens[df_itens['capitulo'] == '99']
```

	descricao_limpa	capitulo	posicao	subposicao	item	subitem
4441889	x	99	99	99	9	9
4337496	seamer drone importado	99	99	99	9	9
4441889	x	99	99	99	9	9
4337496	seamer drone importado	99	99	99	9	9

Fig. 64 - descrição dos itens do capítulo 99

- Da Tabela 1, podemos concluir que a acurácia com menor variação foi a do capítulo, variando entre 94,26% e 94,95%.

- A acurácia do capítulo, retirando-se as stopwords, melhorou ligeiramente em relação à base com stopwords (94,95% sem stopwords contra 94,70% com stopwords). Contudo, quando aplicamos o modelo à base de dados toda, observamos que o desempenho foi pior (20 capítulos com erros e 145 erros no total - com stopwords -, contra 33 capítulos com erros e 210 erros no total - sem stopwords). Provavelmente o modelo sem stopwords teve mais overfitting do que o modelo completo.

- Observamos que a retirada das stopwords e a realização do stemming piorou os resultados de todas as saídas. Isso evidencia que para o presente trabalho a retirada das stopwords não é fator diferencial para melhorar a precisão do modelo uma vez que o texto das descrições são compostos mais por substantivos e adjetivos.

- Por fim, conclui-se que o melhor dos modelos (descrição limpa) teve uma assertividade de 76,87% (23,13% de erros) para classificar o código NCM como um todo. Contudo, observando a assertividade para os dois primeiros dígitos, Capítulo, que foi de 94,70%, podemos concluir que o modelo é bem preciso para prever o Capítulo.

- Dessa forma, sugere-se que o presente trabalho seja um passo inicial para outros trabalhos que tentem resolver o problema de como obter uma acurácia melhor ou igual na classificação dos outros dígitos da NCM (posição, subposição, item e subitem).

- Outra sugestão seria treinar o modelo somente com a base TEC, que é a fonte gabarito dos dados, sem realizar stemming e sem retirar as stopwords.

- Ou, pode-se tentar buscar normalizar a base de descrições buscando uma distribuição uniforme para as categorias da coluna

7. Links

- Notebooks e Relatório

https://github.com/adrianocastanheira/tcc_pucmg

- Vídeo de apresentação

https://youtu.be/xUEHncBjpbo