PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Paloma Castrioto Ribeiro

Victor de Simone Oliveira

Aplicação de Data Science para classificação automática de mensagens em um processo intensivo em conhecimento através de mineração de textos

Orientadora: Fernanda Baião

Co-orientador: Pedro Henrique Piccoli Richetti

Rio de Janeiro
Junho de 2020

AGRADECIMENTOS

À nossa orientadora Fernanda Baião por todo apoio e dedicação ao nosso projeto e ao Pedro Richetti, pela ajuda e suporte principalmente na etapa inicial do trabalho.

Aos nossos pais Soraia, Leonardo, Stela e Marco, por todo o apoio e carinho que sempre nos foi dado.

Aos amigos, que sempre estiveram ao nosso lado nos momentos bons e ruins, por darem mais cor às nossas vidas.

A Catharina Castrioto, Júlia Mendonça e todos que direta ou indiretamente também fizeram parte da nossa formação. Muito obrigado.

RESUMO

A Gestão de Processos de Negócios, ou *Business Process Management* (BPM), tem sido bastante utilizada no mercado de trabalho e estudada no meio acadêmico com o objetivo de identificar gargalos e otimizar processos de negócio. No entanto, cada vez mais esses processos são realizados por empresas e clientes interconectados em rede através das mídias sociais, originando uma quantidade de informações (estruturadas ou não, como documentos e mensagens) sem precedentes. Dentro deste contexto, são usadas técnicas de Ciência de Dados (*Data Science*) com o objetivo de extrair conhecimento útil das informações geradas, sendo o maior foco recente em processos pouco estruturados ou Processos Intensivos em Conhecimento (*Knowledge-intensive Processes* - KiP), que ainda têm suporte inadequado das metodologias e ferramentas existentes, e vêm se estabelecendo como os processos mais críticos nas organizações. Os KiPs envolvem tomadas de decisão complexas que dependem do conhecimento dos participantes do processo, além de serem flexíveis e apresentaram grande variabilidade. Diante deste cenário, este trabalho realiza uma revisão da teoria de KiP e de Teoria dos Atos de Fala, juntamente com as principais técnicas de Processamento de Linguagem Natural (*Natural Language Processing* - NLP), além da criação de um algoritmo em Python que implementa essas técnicas para extrair conhecimento útil das mensagens trocadas entre os participantes de um KiP. Mais especificamente, o objetivo desse algoritmo é realizar a classificação automática de mensagens em categorias de Atos de Fala. Essa classificação pode futuramente servir de base para identificação de padrões de diálogo. O algoritmo criado tem 6 parâmetros, que foram combinados de maneiras distintas para gerar 74 modelos, durante um experimento de Ciência de Dados. Para medir o desempenho de cada um deles, foram utilizadas duas métricas: *accuracy* e *hamming loss.* O modelo que apresentou o melhor desempenho teve uma *accuracy* de 83,35% e um *hamming loss* de 1,17%. Esses resultados foram bem satisfatórios, considerando que este é um problema de NLP *multi-label* com 24 categorias diferentes.

PALAVRAS-CHAVE: Mineração de Textos. Processamento de Linguagem Natural. Data Science. Machine Learning. Business Process Management. Processos Intensivos em Conhecimento. Teoria dos Atos de Fala.

ABSTRACT

Business Process Management (BPM) has been widely used in the companies and studied in the academic environment with the aim of identifying obstacles and optimizing business processes. However, more and more often these processes are carried out by companies and customers interconnected in a network through social media, originating an unprecedented amount of information (structured or not, such as documents and messages). Within this context, Data Science techniques are used in order to extract useful knowledge from the information generated, being the recent focus on poorly structured processes, or Knowledge-intensive Processes (KiPs), which still have inadequate support from existing methodologies and tools, and have been establishing themselves as the most critical processes in organizations. KiPs involve complex decision making that depends on the knowledge of the participants in the process, in addition to being flexible and showing great variability. Given this scenario, this work performs a review of the KiP theory and the Theory of Speech Acts, together with the main techniques of Natural Language Processing (NLP), and describes the creation of an algorithm written in Python that implements these techniques to extract useful knowledge from the messages exchanged between the participants of a KiP in a real scenario. More specifically, the purpose of this algorithm is to perform the automatic classification of messages in categories of Speech Acts, which may in the future serve as a basis for identifying patterns of dialogue. The created algorithm has 6 parameters, which were combined in different ways to generate 74 models, during a Data Science experiment. To measure the performance of each model, two metrics were used: accuracy and hamming loss. The model with the best performance had an accuracy of 83.35% and a hamming loss of 1.17%. These results were quite satisfactory, considering that this is a multi-label NLP problem with 24 different categories.

KEYWORDS: Text Mining. Natural Language Processing. Data Science. Machine Learning. Business Process Management. Knowledge-intensive Processes. Theory of Speech Acts.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Atualmente, vê-se a gestão de processos de negócio sendo investigada no meio acadêmico e científico, e aplicada com sucesso nas empresas, através da proposta e utilização de metodologias, técnicas e ferramentas de BPM (*Business Process Management*) bem estabelecidas. A implementação do BPM busca, entre outros fatores, detectar os gargalos e aperfeiçoar os processos de uma companhia.

Em paralelo, vê-se a gestão de processos não estruturados (KiPs - *Knowledge-intensive Processes)* sendo um grande desafio enfrentado pelas empresas atualmente, uma vez que este tipo de processo tem se tornado cada vez mais crítico nas organizações atuais, fortemente baseadas no conhecimento de seus participantes. Pode-se citar como exemplos de KiPs um atendimento de uma consulta médica, a gestão de uma cabine de controle de uma aeronave e a gestão de incidentes de uma empresa, sendo este último tratado neste trabalho. Esses processos não conseguem ser bem suportados pelas ferramentas de BPM existentes atualmente, como por exemplo, os sistemas BPMS (*Business Process Management System).*

Além disso, é possível ver a tendência crescente do uso de abordagens analíticas (*Data Analytics*) na gestão de empresas, devido à grande quantidade de dados disponíveis. Entretanto, mesmo que parte desses dados sejam compostos por bases estruturadas, boa parte deles podem ser classificados como não estruturados, como repositório de documentos e trocas de mensagens entre usuários de uma rede social. Contudo, esses dados são de extrema importância para as empresas, por terem o potencial de agregar valor à gestão de KiPs, uma vez que refletem a colaboração entre participantes do processo e o conhecimento usado por trás das decisões de negócio. Dentro deste cenário, as técnicas de Ciência de Dados (*Data Science*) possuem como objetivo extrair algum conhecimento útil a partir de um grande volume de dados, que podem ser estruturados ou não. No primeiro caso, são usadas técnicas de *Machine Learning* tradicionais. Já no segundo, são utilizadas técnicas de Mineração de Texto e de Processamento de Linguagem Natural.

Portanto, um dos principais desafios das empresas é definir metodologias e técnicas de BPM para gestão de KiPs, de forma que possam utilizar essa grande quantidade de dados disponível a seu favor através da aplicação de técnicas de *Data Science.* Richetti et al. (2017) propõem em seu trabalho um conjunto de métricas e

indicadores de desempenho que visam suportar a gestão de KiPs, que incluem a perspectiva de comunicação entre os participantes do processo. Essas métricas fazem referência aos atos de fala definidos por Bach e Harnish (1979), que foram extraídos das conversas entre os participantes. Contudo, para obter essas métricas de comunicação, é necessário analisar as interações entre os participantes do processo a partir de conjuntos de dados previamente classificados, tarefa que é bastante complexa, custosa e passível de erros, quando realizada de forma manual.

Dentro de todo este contexto, o objetivo do presente trabalho é a aplicação de uma metodologia de Data Science, incluindo a criação de um algoritmo que consiga classificar sentenças (que se encontram em linguagem natural) em categorias correspondentes a atos de fala específicos, através da utilização de técnicas de Processamento de Linguagem Natural e Mineração de Textos. O algoritmo foi implementado na linguagem Python. O conjunto de dados que será utilizado como base para o aprendizado do modelo de classificação diz respeito à conversas geradas em um KiP de uma empresa real, que atua no ramo de prestação de serviços de TI. Além da criação do algoritmo e o estudo completo da forma que foi construído e seus resultados, será também feita uma revisão bibliográfica de alguns conceitos importantes para a compreensão do projeto como um todo.

O presente estudo está organizado em cinco capítulos. O segundo capítulo diz respeito à todas as bases teóricas necessárias para a compreensão do estudo de caso realizado e suas motivações. Nele, serão apresentadas definições conceituais, revisão histórica e aplicações de processos intensivos em conhecimentos (KiPs) e dos atos de fala que serão usados no estudo de caso. Além disso, também serão apresentados aplicações e técnicas de processamento de linguagem natural, assim como de mineração de texto e *Data Science*. No terceiro capítulo do trabalho será apresentado o estudo de caso em si, e a aplicação da metodologia de Data Science em um cenário real. Primeiramente, será detalhado o processo não estruturado que deu origem ao conjunto de dados, assim como a forma que estes dados foram tratados. Em seguida, o algoritmo construído será apresentado detalhadamente, juntamente às métricas utilizadas e aos testes feitos. Em seguida, no capítulo quatro serão vistos os resultados obtidos com os testes do algoritmo, a partir da comparação do desempenho das métricas obtidas com diferentes parâmetros. Por fim, o capítulo

cinco conclui o trabalho, analisando os resultados da aplicação feita e apresentando direções para possíveis trabalhos futuros.

2. REVISÃO BIBLIOGRÁFICA

Nesta seção serão mencionados alguns conceitos e teorias importantes para a compreensão do trabalho, assim como aplicações e possíveis linguagens de programação para o processamento de linguagem natural.

2.1 Processos intensivos em conhecimento

Processos Intensivos em Conhecimento (*Knowledge-intensive Processes – KiP*) podem ser definidos como processos cuja execução depende fortemente de trabalhadores do conhecimento que realizam várias tarefas de tomada de decisão interconectadas (Vaculin et al., 2011, apud Gonçalves, 2018). Pode-se falar que um processo intensivo em conhecimento é tipicamente colaborativo, orientado por objetivos e depende um processo formal ou informal de comunicação efetiva (Santoro e Baião, 2018). Além disso, segundo Gronau e Weber (2004), os KIPs podem ser caracterizados por um fluxo dinâmico e instável de atividades complexas, que está em constante mudança (apud Gonçalves, 2018).

Na figura 2 podemos ver o espectro de classificação de KIPs, definido por Di Ciccio et al. (2014). Como apresentado, existe uma grande variabilidade de classificações de KiPs que variam em nível de complexidade e flexibilidade. Desta forma, a um processo é atribuído um nível de intensidade de conhecimento, e não uma classificação se é ou não “intensivo em conhecimento”. Esse nível de intensidade em conhecimento pode variar desde processos completamente estruturados (não intensivos em conhecimento), até completamente não estruturados (com alto nível de intensidade em conhecimento), como visto na figura 1.

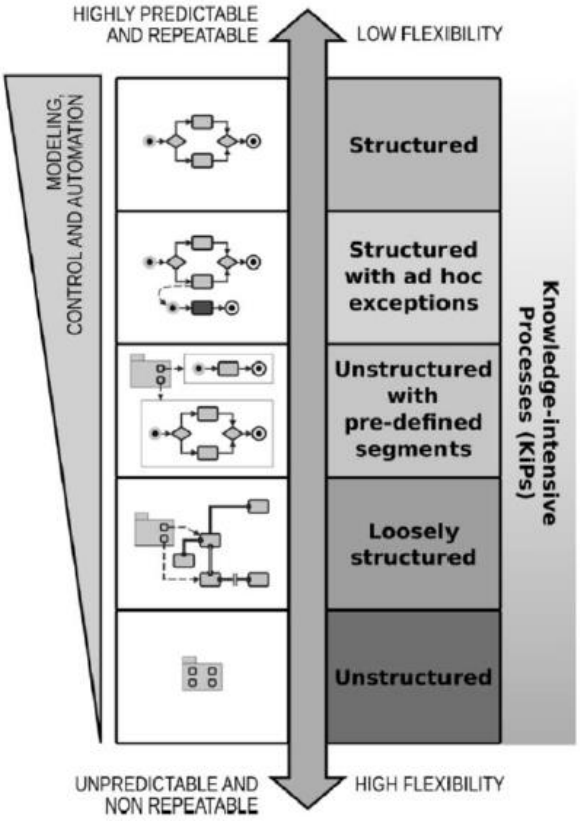

Figura 1 - Classificação de KIP's

Fonte: Di Ciccio et al., 2014

Também, o envolvimento humano e o conhecimento de cada um no processo são essenciais para a execução do KiP, pois é a partir da troca entre os participantes que é possível atingir o objetivo do processo. Contudo, segundo Isik et al. (2013), cada um desses trabalhadores possui seus próprios sentimentos, crenças, intenções e desejos, que influenciam a forma pela qual ele se comunica e interage com os outros. Normalmente, essa interação é feita com o uso de linguagem natural (apud RICHETTI et al., 2017).

Desta forma, os KIP's possuem uma forte colaboração entre os executores, que é frequentemente feita através de mensagens que ficam registradas ao longo do processo de execução. Para compreender melhor a comunicação entre eles, neste trabalho propomos que sejam usadas técnicas de mineração de textos e o Processamento de Linguagem Natural.

2.2 Teoria dos Atos de Fala

Os Atos de Fala podem ser considerados como enunciados produzidos não apenas para descrever estados de coisas, mas também para executar ações que têm como objetivo atingir algum efeito no ouvinte. A teoria é muito usada nos dias de hoje não apenas na área de *Data Science*, como será estudado neste trabalho, mas também na área do direito para a construção e aplicação de leis, por exemplo.

A teoria foi proposta pelo inglês John Langshaw Austin (1962), que entendeu a necessidade de olhar além do significado literal dos enunciados e compreender também como o contexto e a intenção do orador influenciam nele. Segundo a sua teoria, os atos de fala devem ser analisados em três dimensões. O primeiro deles é ato locucionário, que diz respeito ao enunciado em si, ao ato de dizer a frase. O segundo é o ato ilocucionário, que é a intenção do orador com o enunciado. Nesse caso, ao dizer “você está falando muito alto”, não há a intenção apenas de constatar uma situação, mas sim de advertir o orador. Por fim há o terceiro ato, o perlocucionário, que diz respeito ao efeito efetivo do enunciado no ouvinte. Na situação descrita, seria levar o ouvinte a falar mais baixo (apud RICHETTI et al., 2017).

Alguns anos depois, John Searle desenvolveu sua própria classificação dos atos ilocucionários em atos constatativos e performativos. A principal diferença entre as duas categorias é que em uma as sentenças são usadas para descrever fatos e eventos (constatativos) e na outra elas são usadas para realizar uma ação (performativos) (MARCONDES, 2006). Em sua obra “A Taxonomy of Speech Acts” Searle (1979) definiu um ato ilocucionário como a menor unidade possível da comunicação humana, como por exemplo comandos e perguntas (apud RICHETTI et al., 2017). Searle (1979) propôs cinco classes principais nas quais os atos ilocucionários se dividem:

- Assertivos ou representativos: mostram a crença do locutor quanto à verdade de uma proposição. Um exemplo seria: “eu faço aniversário em julho”.
- Diretivos: possuem como objetivo gerar uma ação específica no ouvinte. Logo, um exemplo possível é a pergunta “poderia me passar o sal?”
- Compromissivo ou comissivos: comprometem o locutor com uma ação futura. Um exemplo é a frase “te ligarei hoje à noite”.
- Expressivos: expressam as atitudes e emoções do locutor. Por exemplo, “meus pêsames pela sua perda”.

- Declarativos: produzem uma nova situação no mundo. Um exemplo é um padre em um casamento, ao falar a frase “eu os declaro marido e mulher”.

As classificações dos atos de fala foram expandidas por Bach e Harnish (1979), que sugeriram novas classificações a partir da divisão de Searle. Na figura 2, pode-se encontrar todas as classificações de Bach e Harnish (1979). Os atos de fala constatativos se encontram no primeiro grupo da figura (Constatives), enquanto os atos de fala performativos são subdivididos nos outros três grupos da figura, que direcionam alguma ação quando ditos.

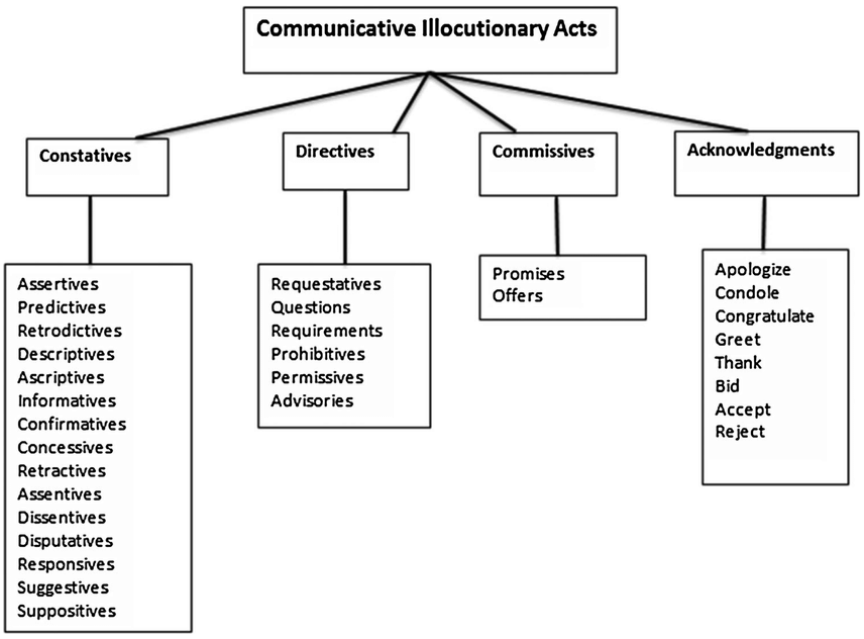

Figura 2 - Classificação de Atos Ilocucionários de Bach e Harnish

Fonte: Bach e Harnish (1979)

As classificações dos atos de fala de Bach e Harnish (1979) serão usadas para classificar o dataset usado no estudo deste trabalho. Podemos definir cada grupo dos atos ilocucionários como:

- Constatativos (*constatives*): quando a sentença traz algum ponto de vista do locutor. Exemplo: “você é bonita”.

- Diretivos (*directives*): quando a sentença solicita do ouvinte que diga ou faça alguma coisa. Exemplo: “que horas vamos sair?”.

- Compromissivos (*comissives*): quando o locutor se compromete em relação a uma ação futura. Exemplo: “eu prometo te encontrar amanhã”.

- Expressivo (*acknowledgments*): quando a sentença revela o estado de espiríto ou psicológico do locutor. Exemplo: “muito obrigada pelo presente!”.

2.3 Processamento de linguagem natural

Segundo Brownlee (2017), o Processamento de Linguagem Natural ou *Natural Language Processing* (*NLP*) pode ser definido como a manipulação automática de linguagem natural, como fala e texto, por programas de computador.

O primeiro passo fundamental de um algoritmo de NLP é a identificação de cada palavra em um texto, seguida de sua caracterização, que é feita através de diversas etapas e técnicas. Uma delas, por exemplo, é o *Part-of-speech Tagging* (POS Tagging), que é o ato de conferir uma classe gramatical a uma palavra. Outras técnicas, que serão descritas mais em detalhes neste capítulo, são tokenização, nuvem de palavras, *dependency parsing, stemming* e *lemmatization*.

Nesta seção, introduziremos conceitos importantes relacionados ao NLP, trazendo à tona os principais campos onde é usado, referências na literatura e as principais técnicas utilizadas.

2.3.1 Aplicações

Existem diversas aplicações do NLP atualmente, encontradas nas mais diversas áreas de conhecimento. Grande parte dessas aplicações são muito presentes no cotidiano de todos nos dias de hoje.

Um primeiro exemplo são os assistentes virtuais (*chatbots)*, que são programas de computador que simulam uma conversa com um ser humano por meio de aplicativos de conversa – como *Whatsapp* ou *Facebook Messenger* – em linguagem natural. A utilização de *chatbots* é muito presente hoje em dia em empresas de venda *on-line*, como a Magazine Luiza por exemplo. Essas companhias utilizam a ferramenta como forma de interagir com (potenciais) clientes automaticamente, tratando algumas de suas solicitações frequentes de forma rápida e atenciosa. Na figura 3 encontra-se um exemplo do *chatbot* da empresa Magazine Luiza.

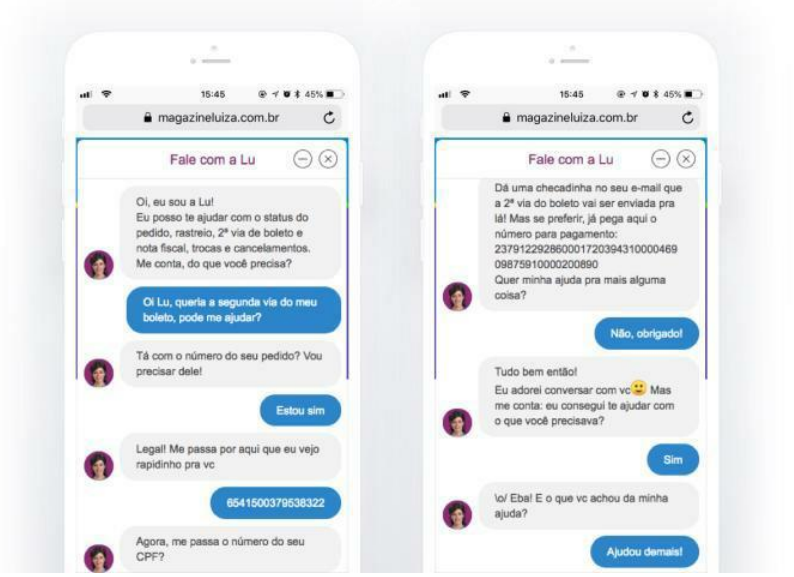

Figura 3 - Exemplo do chatbot sendo usado no site da Magazine Luiza

Fonte: https://simple.nama.ai/post/lu-o-chatbot-da-magazine-luiza-que-e-queridinho-do-publico

Os chatbots utilizam técnicas de NLP para seu desenvolvimento, uma vez que o “robô” (i.e., o algoritmo do assistente virtual) precisa processar cada mensagem em linguagem natural recebida do cliente e elaborar mensagens para responder suas perguntas, também em linguagem natural.

Outra aplicação do NLP presente no cotidiano de muitas pessoas são os assistentes pessoais, encontrados nos *smartphones*, *tablets* e computadores, como a Siri (Apple) e a Bixby (Samsung). Esses programas usam a Inteligência Artificial para reconhecimento de comandos de voz elaborados em linguagem natural e, então, realizam atividades de acordo com o desejo do usuário.

Além disso, as técnicas de processamento de linguagem natural também são usadas por aplicativos de tradução simultânea, como o *Google Translator*.

2.3.2 Representações de palavras

O grande diferencial dos dados em linguagem natural é que eles são não estruturados, ou seja, não existem atributos ou classes numéricas isolados e definidos *a priori*. Para que seja possível criar um modelo computacional com o objetivo de realizar predições, é preciso que esses dados sejam transformados em dados estruturados, numéricos ou categóricos. Essa conversão é o que chamamos de

“representações de palavras”, ou seja, busca-se representar as palavras através de números.

A seguir será apresentada a representação conhecida como conjunto de palavras (*Bag of Words* - BoW), uma das mais utilizadas e simples, e que será adotada para este trabalho. Para exemplificar, considere três frases:

Frase 1: eu não gostei desse filme

Frase 2: esse filme é muito bom

Frase 3: gostei muito

A ideia é representar cada palavra nesse conjunto de frases como um atributo, ou seja, uma coluna de uma matriz. Essa matriz ficaria como na Tabela 1.

Tabela 1 - Exemplo de *Bag of Words*

	eu	não	gostei	desse	filme	esse	é	muito	bom
Frase 1	1	1	1	1	1	0	0	0	0
Frase 2	0	0	0	0	1	1	1	1	1
Frase 3	0	0	1	0	0	0	0	1	0

Fonte: Elaborado pelos autores

Dessa maneira, consegue-se representar uma frase (ou mensagem) por um vetor de zeros e uns. Por exemplo, a frase 1 seria representada pelo vetor (1, 1, 1, 1, 1, 0, 0, 0, 0). Outras representações também consideram a frequência, para casos em que um mesmo termo aparece mais de uma vez em uma mesma frase.

A seguir, serão mostradas diversas técnicas de NLP que tem o poder de tratar os dados de entrada para que se possa criar um modelo melhor.

2.3.3 Técnicas de NLP

Em se tratando de algoritmos que se utilizem de técnicas de NLP, é fundamental fazer com que o computador entenda o que uma frase em linguagem natural quer dizer. Aborda-se abaixo algumas das mais fundamentais.

A tokenização[1] é o primeiro passo fundamental no processamento de dados não estruturados. Segundo Manning et. al (2008), tokenização é a tarefa de separar uma cadeia de texto em pedaços, chamados de *tokens*, muitas vezes ignorando pontuações. De maneira geral, esses *tokens* serão as palavras dessa cadeia de texto.

Muitas vezes, depois de identificadas todas as palavras em um texto através da tokenização, é construída uma visualização denominada de “nuvem de palavras”, com o objetivo de identificar quais são os termos mais frequentes. A Figura 4 ilustra um exemplo:

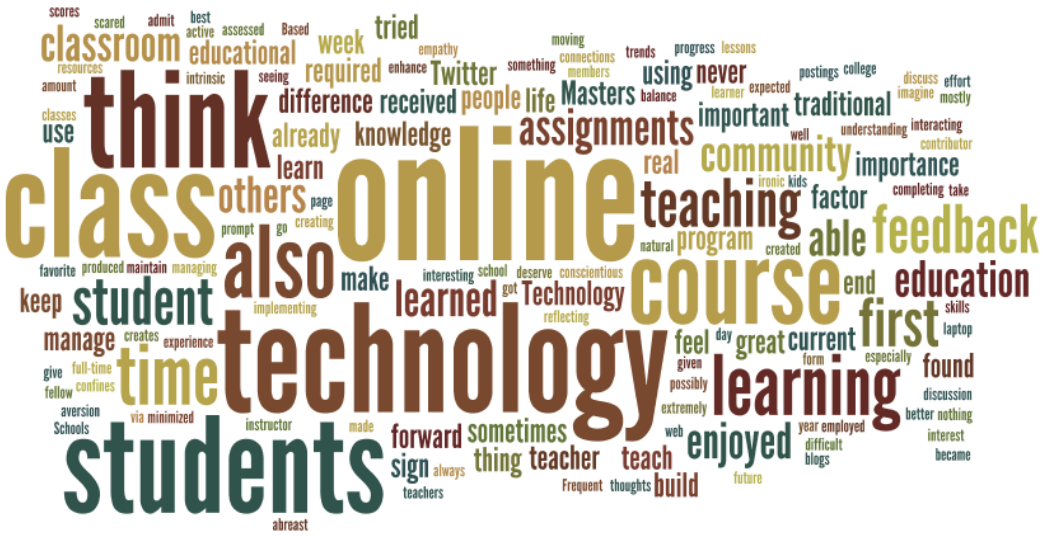

Figura 4 - Ilustração de uma nuvem de palavras

Fonte: https://br.pinterest.com/pin/481744491363086469/

Uma técnica sempre utilizada em algoritmos de NLP é a de remoção de *stopwords*. Segundo Raujli e Saini (2016), *stopwords* são palavras extremamente comuns, encontradas em qualquer linguagem natural que carregam pouco ou nenhum contexto semântico em uma frase. Exemplos comuns de *stopwords* incluem artigos e preposições. É uma operação de pré-processamento necessária para facilitar o processamento de texto.

[1] https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html

Uma outra técnica extremamente comum é a *part-of-speech* (POS) *tagging*. Segundo Jackov (2015), é a tarefa de rotular as palavras de um texto descrevendo as suas várias características gramaticais, principalmente sua classe gramatical.

Complementando o *POS tagging*, o *dependency parsing* é uma técnica que descreve a função sintática de uma palavra em uma oração (Kadam, 2019). A figura 5 ilustra uma imagem gerada pela função *displacy* da biblioteca do Python *SpaCy*, que exemplifica a técnica *dependency parsing*.

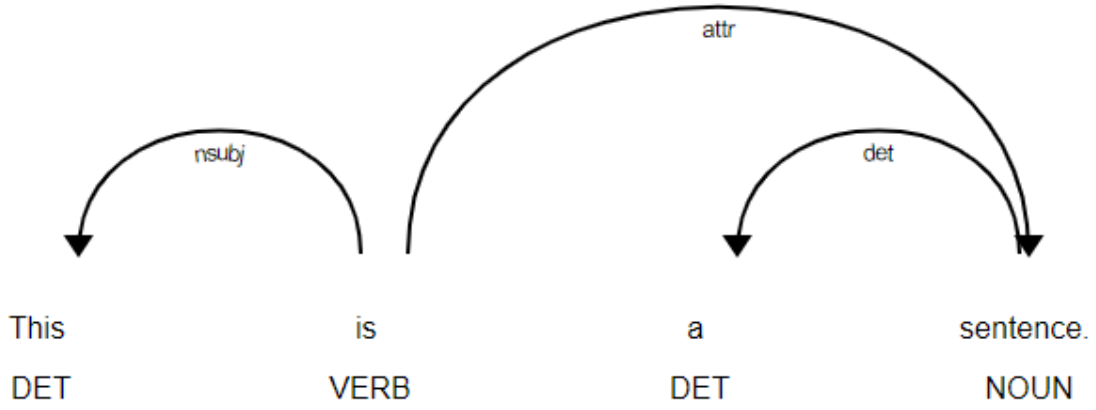

Figura 5 - Ilustração da técnica *dependency parsing* sobre uma frase

Fonte: https://spacy.io/usage/visualizers

As técnicas mencionadas anteriormente são de identificação e caracterização das palavras existentes em um texto. Aprofundando um pouco mais, existem outras técnicas mais avançadas de NLP que visam reduzir o número de palavras e manter apenas os termos que representam a sua essência. Muitas vezes, é desejável identificar, por exemplo, que “andar” e “andando” são palavras derivadas de um termo comum (desinências de um mesmo verbo, “andar”), e que transmitem a mesma ideia. Para isso, existem duas técnicas de normalização de textos muito utilizadas: *stemming* e *lemmatization*.

Stemming é a abordagem mais simples das duas. Sua ideia principal é reduzir as palavras à sua raiz (ou radical), que é uma forma reduzida/truncada de uma palavra. Por exemplo, a raiz das palavras “andar” e “andei” é “and-”, que também é a raiz de outras variações como “andando”, “andado”, “andou”, etc.

Lemmatization, por sua vez, é uma técnica bem mais sofisticada, que se utiliza da técnica *POS tagging* para reduzir as palavras ao seu lema, que é a forma

deflexionada de uma palavra. Por exemplo, o lema das palavras “andar” e “andei” é “anda” (diferentemente do *stemming* que realiza simplesmente um truncamento da palavra). Isso permite reduções mais sofisticadas que não são possíveis com o *stemming*, como das palavras “fui” e “será”, ambas possuindo o mesmo lema, “ser”.

2.3.4 Linguagens e bibliotecas

Para construir algoritmos computacionais que implementam técnicas de NLP, utiliza-se alguma linguagem de programação que seja compatível com um ambiente de programação escolhido. Este ambiente de programação é representado por uma ferramenta cujas funcionalidades contemplam, no caso de NLP, a implementação das técnicas descritas na Seção 2.3.3. Existem diversas linguagens com ambientes de programação correspondentes capazes de realizar estas tarefas, sendo as mais frequentemente utilizadas o Python e o R. Neste trabalho, decidimos utilizar o Python devido à familiaridade com a linguagem. Além disso, é importante ressaltar que as técnicas implementadas neste trabalho são adaptadas apenas à língua inglesa.

Existem duas bibliotecas muito abrangentes de NLP em Python: NLTK e *SpaCy* (COSTA, 2020). Ambas têm como finalidade realizar a interpretação de frases escritas em linguagem natural. A NLTK surgiu em 2001, sendo uma biblioteca bem mais antiga e madura, contando com uma grande variedade de métodos e funções, como pode ser no sítio Github. A *SpaCy*, por outro lado, teve sua primeira versão lançada em 2015, sendo uma biblioteca mais recente e que tem como premissa trabalhar apenas com os métodos e funções mais atualizadas e otimizadas. Neste trabalho será feito o uso de ambas bibliotecas.

A NLTK será utilizada para realizar o *stemming.* Segundo Heidenreich (2018), os algoritmos mais conhecidos que desempenham este trabalho são o *PorterStemmer* (PORTER, 1980), *SnowballStemmer* (PORTER, 2001) *e LancasterStemmer* (PAICE, 1990). Cada algoritmo possui o seu “grau de agressividade”, ou seja, determina o quanto ele realiza truncamentos nas palavras. Por exemplo, se for agressivo demais, termos muito diferentes podem acabar com a mesma raiz, como por exemplo “telefone” e “televisão” transformando-se em “tele-”. Por outro lado, se for excessivamente conservador, também gera resultados indesejados, como por exemplo “conectar” transformando-se em “conecta-” e “conectados” transformando-se

em “conectad-”, sendo que é desejável que sejam reduzidas à mesma raiz. Os algoritmos foram apresentados em ordem de grau de agressividade, sendo o *PorterStemmer* o mais conservador e o *LancasterStemmer* o mais agressivo. O *SnowballStemmer*, além de ser uma versão atualizada do *PorterStemmer* (SNOWBALL, 2001), não é nem tão conservador e nem tão agressivo, sendo a opção mais adequada para este trabalho.

Já a biblioteca SpaCy será utilizada para tratar do *lemmatization*. A *SpaCy* conta com uma função *load*, que carrega um vocabulário que contém dezenas de informações sobre as palavras da língua inglesa, como seu lema.

2.4 Mineração de textos

A mineração de textos é um campo de *Data Science* que visa extrair conhecimento útil a partir de dados não estruturados, utilizando-se das representações de palavras e de técnicas de NLP como visto nas seções anteriores, e também de técnicas de mineração (como classificação ou agrupamento), que serão descritas nesta seção. Nos dias de hoje, a mineração de textos é completamente fundamental visto que muitos dos dados obtidos não são estruturados.

2.4.1 Classificadores

O algoritmo a ser implementado neste trabalho trata-se de um algoritmo de classificação. Segundo Leonel (2019), classificação nesse contexto é uma abordagem supervisionada de Machine Learning, na qual o algoritmo aprende através dos dados de entrada alimentados a ele, e, em seguida, usa esse aprendizado para classificar novas observações. Uma abordagem supervisionada assume que os dados de entrada já estão classificados.

Esses problemas de classificação podem ser de 3 tipos: binários, *multi-class* e *multi-label.* Os problemas de classificação binários são caracterizados por existirem apenas dois *labels* possíveis, normalmente “sim” ou “não”. Um exemplo é a classificação de *emails* como sendo *spam* ou não. Os problemas *multi-class* são aqueles que apresentam mais de dois *labels* possíveis. Um exemplo é a classificação de espécies de planta. Finalmente, os problemas *multi-label* são aqueles que apresentam mais de dois *labels* possíveis, onde um mesmo exemplo pode pertencer

a um ou mais *labels*. Um exemplo é o reconhecimento de objetos em uma foto, onde podem existir múltiplos objetos, como “carro”, “bicicleta”, “pessoa”, etc. (BROWNLEE, 2020).

Em se tratando dos classificadores, que têm como objetivo criar e ajustar o modelo de classificação, existem dezenas de opções, e para este trabalho trazemos três: *Gaussian Naive Bayes (GaussianNB)*, *Support Vector Machine* (SVM) e *LogisticRegression*.

Segundo Schultebraucks (2017), *GaussianNB* é um classificador que trabalha baseando-se no Teorema de Bayes, que descreve a probabilidade de um evento baseada no conhecimento das condições de eventos relacionados a este. Ele possui *naive* no nome, do inglês “ingênuo”, pois considera que não existe correlação entre dois *labels* (classes) (SCIKIT-LEARN, 2019).

O classificador SVM utiliza-se da ideia que os dados estão alocados em um espaço de *n* dimensões, onde cada eixo representa uma dimensão. A ideia do SVM é encontrar o “hiperplano” que separa as amostras de classes diferentes nesse espaço *n*-dimensional. No caso de ter-se apenas 2 dimensões, encontra-se a reta que separa as duas amostras, como exemplificado na figura 6.

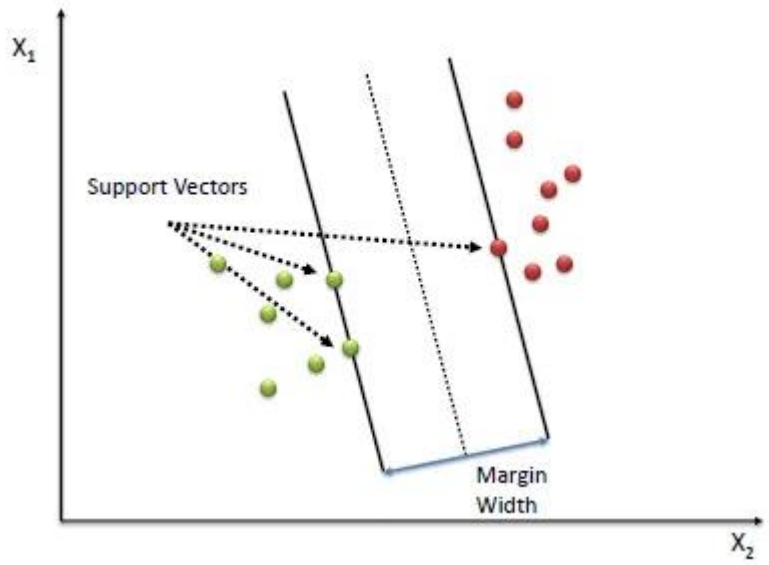

Figura 6 - Ilustração do funcionamento da técnica de classificação SVM com 2 dimensões
Fonte: (WANG, 2018)

Inicia-se com uma reta (que pode ser definida aleatoriamente, ou através de uma função) e, em seguida, encontra-se a amostra de cada classe mais próxima da

desta reta. Estas amostras são denominadas vetores de suporte, ou *support vectors*. O objetivo do SVM é maximizar a distância entre esses vetores e a reta (hiperplano). Aqui é importante notar que utilizaremos um caso especial do SVM, o SVC (*Support Vector Classifier*), que trata de problemas de classificação (WANG, 2018).

Finalmente, segundo Brownlee (2018) o *LogisticRegression* tem seu nome devido à função utilizada no coração do método, a função logística. Esta função, também denominada função sigmóide, foi desenvolvida por estatísticos para descrever propriedades do aumento de população em ecologia, que tem um comportamento de rápido crescimento e atinge um valor máximo na capacidade de alocamento do ambiente. É uma curva em forma de S que pega qualquer valor real e transforma em um valor entre 0 e 1, mas nunca chega exatamente nesses limites. A fórmula desta função é mostrada na equação 1:

$$f(x) = 1 / (1 + e^{-x}) \qquad (1)$$

Onde x é o valor de entrada e f(x) o valor de saída, entre 0 e 1. A regressão logística utiliza uma equação como representação, mostrada na equação 2:

$$y = e^{(b0 + b1*x)} / (1 + e^{(b0 + b1*x)}) \qquad (2)$$

Esta função é para o caso onde existe apenas uma variável explicativa no modelo. Nesse caso, b0 é o intercepto da função e b1 é o coeficiente desta única variável. Cada coluna do *dataset*, isto é, cada variável explicativa, é associada a um parâmetro “b”, a ser ajustado através dos dados de entrada. “y” é o valor de saída, entre 0 e 1, como visto para a função logística anteriormente. O objetivo da equação, portanto, é gerar um valor entre 0 e 1 baseado nos valores das variáveis explicativas. Depois de obter-se esse valor, realiza-se um arredondamento. Por exemplo, se o valor de y = 0,345, tem-se que y = 0. Conclui-se que a regressão logística retorna uma saída binária, sendo amplamente utilizado para problemas de classificação binária.

2.4.2 Métodos de transformação de problemas

Como visto na seção anterior, problemas de classificação podem ser do tipo multi-label, que é bem complexo, devendo-se adotar um método de divisão do problema maior em diversos subproblemas mais simples. Aqui abordaremos dois deles: *Binary Relevance* e *Label Powerset*.

Segundo Luaces et. al (2012), o *Binary Relevance* decompõe o aprendizado de um problema em um conjunto de subproblemas de classificação binários, um para cada *label*, onde cada modelo associado a esse subproblema é aprendido independentemente, usando apenas a informação daquele *label* em particular e ignorando as informações de todos os outros.

O *Label Powerset*, por outro lado, adota uma estratégia diferente. Esse método transforma o problema em um problema do tipo multiclasses, onde as variáveis explicativas são combinadas e cada combinação é tratada como uma classe única (SZYMANSKI, 2017).

3. METODOLOGIA

Nas próximas seções será apresentado o estudo de caso deste trabalho, que constituiu uma aplicação de Data Science para classificação automática de mensagens trocadas entre os participantes de um processo intensivo de conhecimento de Gestão de Incidentes de TI, a partir de dados reais de uma empresa de prestação de serviços de TI. Será mostrado o dataset usado em detalhes, assim como todo o processo de pré-processamento e mineração. Além disso, será visto também como foi feita a criação do modelo e quais parâmetros foram utilizados, assim como os resultados do experimento que avaliou vários cenários.

Importante ressaltar que o código descrito neste trabalho está disponível no seguinte link do Github: https://github.com/vsmoliveira/TFC-NLP-2020.1.

3.1 Descrição do cenário de aplicação

O processo em questão é a gestão de incidentes de uma empresa de prestação de serviços de TI, que se baseia na troca de mensagens entre os seus clientes e o suporte técnico da empresa. Esse processo é baseado no padrão ITIL, que é composto das seguintes etapas (OLIVEIRA, 2017):

- Detecção do incidente: o cliente abre um incidente junto ao suporte técnico da empresa.
- Classificação: é definida a categoria a qual o incidente descrito pelo cliente se encaixa. Além disso, é determinado o nível de urgência.
- Diagnóstico: momento no qual a equipe de TI usufrui de seus conhecimentos para entender mais a fundo o problema trazido pelo cliente, fazendo questionamentos direcionados. Caso o atendendo veja que não consegue solucionar a questão, ele encaminha o incidente para um próximo nível (como o agendamento de suporte presencial, por exemplo).
- Resolução: compreender se a base de conhecimento do atendente do suporte técnico foi suficiente para solucionar a questão trazida pelo cliente. Se não, direcioná-la para o próximo nível.
- Fechamento do incidente: nesta etapa são arquivados todos os detalhes do incidente. Ela é essencial, pois esse conhecimento adquirido no processo muito provavelmente será útil para a resolução de futuros incidentes.

- Monitoramento: pode acontecer de o problema trazido pelo cliente não ser resolvido no primeiro contato com a equipe de suporte. Desta forma, é necessário manter um monitoramento do histórico dos clientes, para que o atendimento seja mais eficaz quando o mesmo contatar a empresa novamente.

Desta forma, por tratar-se de um processo que envolve grande colaboração entre os participantes e que é orientado pelo objetivo comum de resolver uma dúvida ou um problema trazido pelo cliente, pode-se considerá-lo um KiP. Por se tratar de um sistema de comunicação entre os envolvidos solução de problemas, o teor das mensagens encontradas nesse processo encontra-se em linguagem natural. Logo, é possível classificar essas mensagens em determinados atos de fala, usando as técnicas apresentadas na Seção 2.

Tendo isso em mente, o presente trabalho implementou um algoritmo de Machine Learning que pudesse classificar automaticamente as mensagens trocadas durante este processo, utilizando-se de técnicas de NLP. Isso seria muito útil para os gestores pois os possibilitaria a enxergar quais características de mensagens são as mais problemáticas dentro do processo, tendo em vista a otimização do tempo de solução do incidente, e assim poder direcionar o processo para que seja tratado de forma mais eficiente.

3.2 Dataset

3.2.1 Visão geral do dataset

A base de dados deste processo foi disponibilizada no formato de uma planilha Excel, contendo 35.510 entradas com os seguintes atributos:

- ticketid {int}: código identificador do chamado referente à solicitação do atendimento do usuário (ticket)
- msgId {int}: código identificador da mensagem
- msgContent {str}: conteúdo textual da mensagem
- MimirQueryString {Declarative, Informative, Question, Requestive, Descriptive, Advisory, Responsive, Suggestive, Retrodictive, Assertive, Confirmative, Concessive, Permissive, Prohibitive, Suppositive,

Requirement, Expressive, Assentive, Commissive, Retractive, Disputative, Predictive, Dissentive, Ascriptive}: classificação do ato de fala

- timestamp {str}: data e hora de envio da mensagem
- duration {str}: duração do ticket, em horas
- sendertype {customer, agent}: quem enviou a mensagem

Um ponto importante a ser ressaltado é quanto ao campo MimirQueryString, que contém a classificação de cada mensagem nas categorias dos atos de fala apresentados na Seção 2. Esta categorização foi atribuída por um conjunto de regras definido previamente por especialistas do domínio.

É fundamental relembrar que essa base de dados é *multi-label*, ou seja, pode existir mais de uma classificação para uma mesma mensagem.

3.2.2 Pré-processamento

A tabela 2 exibe 11 linhas selecionadas do *dataset* original, compreendendo 7 mensagens diferentes.

Tabela 2 - Conteúdo do *dataset* original

ticketid	msgId	msgContent	MimirQueryString	timestamp	duration	sendertype
240638	765523	You will need to understand this demand with Reinaldo	Advisory	18/12/2015 08:59	121.3	customer
234328	743067	You will need to submit an analyst on site.	Assertive	09/11/2015 12:19	48.5	agent
224557	710997	You will need to check with the technical team a spot te	Confirmative	08/09/2015 16:46	4.5	agent
224557	710997	You will need to check with the technical team a spot te	Declarative	08/09/2015 16:46	4.5	agent
222123	703145	You will be asked to quote Commercial Techmaster.	Requestive	25/08/2015 10:28	33.1	agent
222808	706605	You said it would not be possible to perform remotely a	Requestive	31/08/2015 17:57	125.1	agent
231850	734404	You said it is providing a new ADSL modem and calls the	Declarative	22/10/2015 09:16	7.0	agent
231850	734404	You said it is providing a new ADSL modem and calls the	Descriptive	22/10/2015 09:16	7.0	agent
241056	768188	You reported that DCSRJCC02 server has iis 7 installed.	Declarative	28/12/2015 09:20	163.3	customer
241056	768188	You reported that DCSRJCC02 server has iis 7 installed.	Informative	28/12/2015 09:20	163.3	customer
241056	768188	You reported that DCSRJCC02 server has iis 7 installed.	Retrodictive	28/12/2015 09:20	163.3	customer

Fonte: Elaborado pelos autores

Após uma extensa análise dos dados, foram aplicadas as seguintes etapas de pré processamento:

1) Seleção de características: Retirada dos atributos não essenciais para o modelo, restando apenas msgContent e MimirQueryString.

2) Limpeza de dados: Correção de 16 mensagens com o erro “#NOME” do Excel. Muitas delas iniciavam com “=”, portanto o Excel esperava uma fórmula, o que causou o erro. Assim, foi realizada uma substituição de “=” para “” (vazio), solucionando o problema.

3) Seleção de dados: Depois do procedimento acima, 11 das 16 mensagens ainda exibiam o valor “#NOME”, pois “#NOME” era o conteúdo da mensagem em si. Provavelmente isso ocorreu por algum erro prévio de manipulação dos dados através do Excel. Por causa disso, as 11 entradas referentes a essas mensagens foram descartadas.

4) Binarização: A coluna contendo as classificações da mensagem, ou seja a variável de classe da classificação (MimirQueryString), foi transformada em uma matriz com o mesmo número de linhas e com 24 colunas, cada uma referente à uma categoria (*label*) correspondendo a um tipo de Ato de Fala, e recebendo o nome desta categoria. Todos os valores dessa matriz são binários, ou seja, “1” caso a mensagem da linha em questão tenha como classificação a categoria da coluna em questão, e “0” caso contrário. Essa codificação foi feita para lidar melhor com o problema de *multi-label*.

A tabela 3 exibe 7 linhas selecionadas do *dataset* modificado, compreendendo 7 mensagens diferentes. Diferentemente do *dataset* original, cada existe apenas uma linha para cada mensagem. É importante notar que apenas 6 das 24 colunas da matriz de *features* (todas as colunas menos a “msgContent”) estão sendo exibidas.

Tabela 3 - Extrato da etapa de binarização da variável de classe

msgContent	Declarative	Informative	Question	Requestive	Descriptive	Advisory
You will need to understand this demand with Reinaldo	0	0	0	0	0	1
You will need to submit an analyst on site.	0	0	0	0	0	0
You will need to check with the technical team a spot te	1	0	0	0	0	0
You will be asked to quote Commercial Techmaster.	0	0	0	1	0	0
You said it would not be possible to perform remotely a	0	0	0	1	0	0
You said it is providing a new ADSL modem and calls the	1	0	0	0	1	0
You reported that DCSRJCC02 server has iis 7 installed.	1	1	0	0	0	0

Fonte: Elaborado pelos autores

3.3 O algoritmo

Como visto anteriormente, a linguagem escolhida para a implementação do algoritmo foi o Python. Nesta seção, será exibido o passo a passo dessa implementação, explicitando o código.

3.3.1 Coleta dos dados

A primeira etapa é carregar a base de dados para ser processada no ambiente de programação em Python. Para isto, utiliza-se a função *read_excel* da biblioteca *pandas*, que lê uma aba de um arquivo Excel e guarda as informações contidas nas células dessa aba em uma variável do tipo *dataframe*, que é essencialmente uma tabela. Essa informação foi atribuída à variável *dataset*. Foi também criada uma variável denominada *labels_dataframe*, guarda as mesmas informações que a variável *dataset*, exceto pela coluna de mensagens, ou seja, guarda apenas as informações dos *labels*. O código pode ser visto na figura 7:

```
script_path = os.path.dirname(__file__)
dataset = pd.read_excel(os.path.join(script_path, 'Datasets\Dataset OneHot.xlsx'))
labels_dataframe = dataset.iloc[:, 1:]
```

Figura 7 - Código das variáveis *dataset e labels_dataframe*

Fonte: Elaborado pelos autores

Tabela 4 - Conteúdo da variável *dataset*

Index	msgContent	)eclarative	nformativ	Question	Requestive	)escriptive	Advisory
20	You will need to understand this demand with Reinaldo…	0	0	0	0	0	1
21	You will need to submit an analyst on site.	0	0	0	0	0	0
22	You will need to check with the technical team a spot…	1	0	0	0	0	0
23	You will be asked to quote Commercial Techmaster.	0	0	0	1	0	0
24	You said it would not be possible to perform remotely…	0	0	0	1	0	0
25	You said it is providing a new ADSL modem and calls t…	1	0	0	0	1	0
26	You reported that DCSRJCC02 server has iis 7 installe…	1	1	0	0	0	0

Fonte: Elaborado pelos autores

Pode-se constatar que a variável *dataset* contém exatamente as mesmas informações encontradas no Excel (ver figura 6 da seção anterior). Assim, com a base carregada, pode-se continuar para a etapa de pré-processamento dos dados.

3.3.2 Caracterização dos dados

Antes de prosseguir com o pré-processamento, serão realizadas algumas análises estatísticas sobre os dados do dataset apresentado.

Primeiramente, na figura 8 encontra-se um gráfico de barras que mostra o número de classificações dos diversos Atos de Fala.

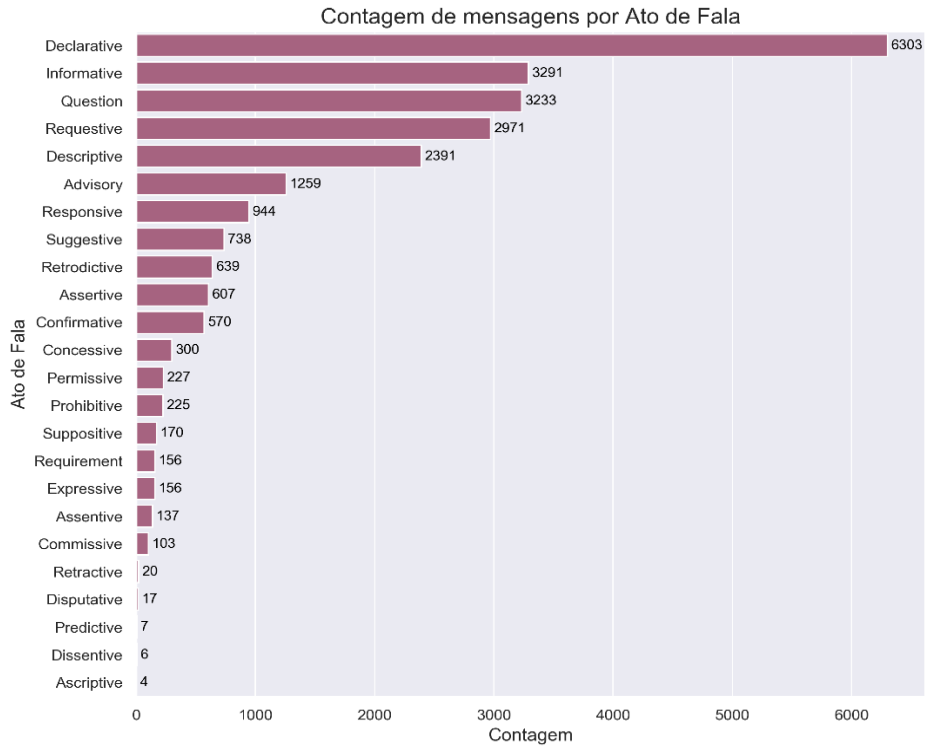

Figura 8 - Quantidade de mensagens classificadas para cada ato de fala

Fonte: Elaborado pelos autores

Percebe-se que os dados são bem desbalanceados, ou seja, o número de classificações não é bem distribuído entre as classes. Em função deste desbalanceamento, cabe ressaltar que um dos cenários avaliados aplicou uma técnica de agrupamento de dados, unindo os subconjuntos das 5 categorias menos frequentes em uma mesma categoria. No entanto, como não houve ganho nos resultados, esta etapa de transformação for desconsiderada no restante do experimento.

Visto que o dataset do estudo de caso apresenta *multi-labels*, outra informação relevante de ser apresentada é o número de mensagens por quantidade de labels, como visto no gráfico representado na figura 9. A partir dele é possível concluir que a

maioria das sentenças (7.993) apresenta apenas um label, enquanto o resto apresenta de 2 a 6.

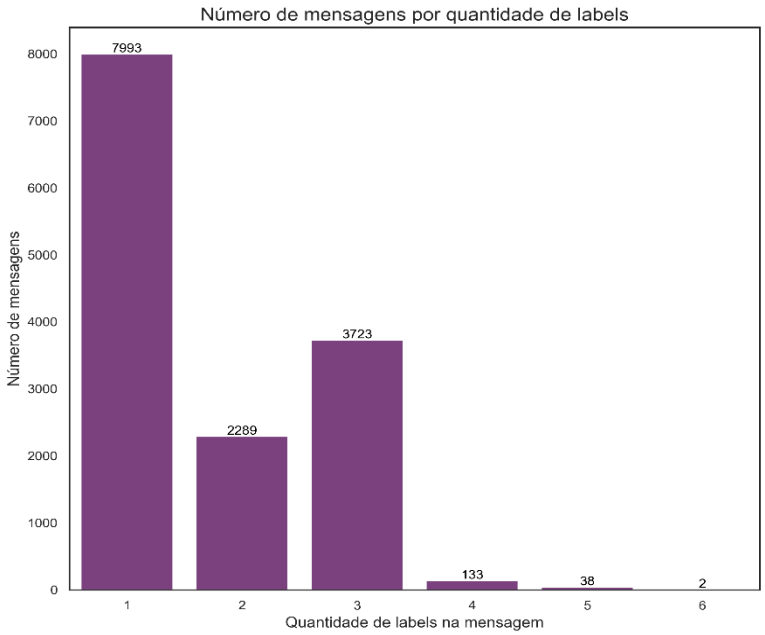

Figura 9 - Quantidade de mensagens por número de labels

Fonte: Elaborado pelos autores

Além disso, foi analisado o tamanho das sentenças do dataset. O histograma representado na figura 10 mostra a quantidade de mensagens distribuída pelo tamanho delas, em quantidade de caracteres.

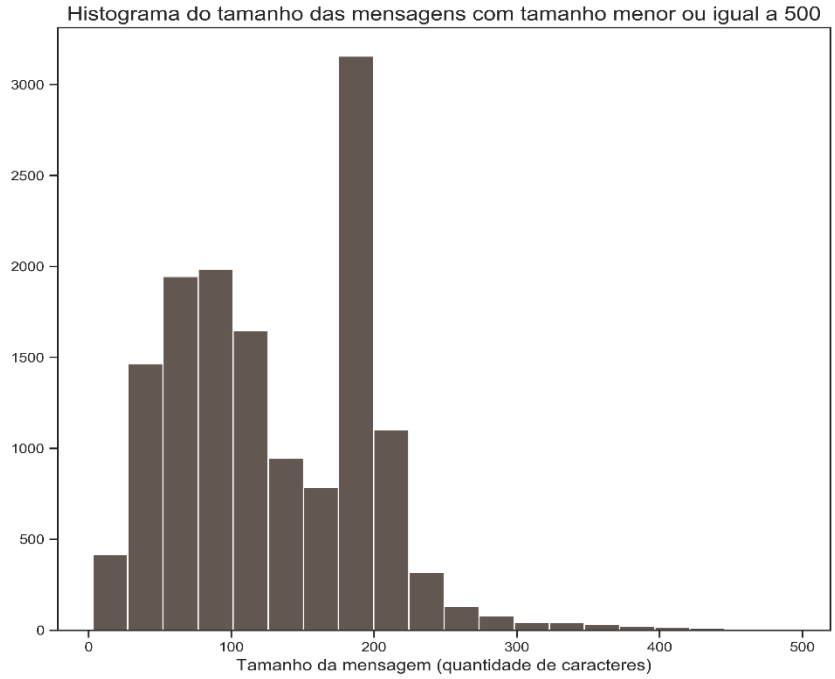

Figura 10 - Quantidade de mensagens por número de caracteres

Fonte: Elaborado pelos autores

Logo, conclui-se que grande parte das mensagens possui de 175 a 200 caracteres de extensão. É importante ressaltar que este histograma está fazendo um corte em 500 caracteres, para facilitar a visualização dos dados, visto que a amostra de mensagens que possuem mais de 500 caracteres é extremamente baixa.

3.3.3 Pré-processamento dos dados

Em primeiro lugar, antes de passar para a fase de aprendizado do modelo, é necessário tratar os dados coletados. Este é um passo fundamental do processo de Data Science e da Mineração de Textos em especial, tendo em vista que existem diversas questões dos dados não estruturados que precisam ser tratadas antes de aplicar as técnicas de mineração de textos (por exemplo, muitos caracteres e palavras indesejadas nos dados de entrada). É nessa parte, por exemplo, que as mensagens são reduzidas para conter apenas as palavras que são essenciais para que o algoritmo consiga fazer as associações adequadas.

Para isso, primeiro é realizada uma limpeza geral das mensagens. Números, pontuações, espaços desnecessários e caracteres especiais como “#”, “@” e “$” são

caracteres indesejáveis neste caso. Além disso, não é desejável que haja diferenciação entre palavras que são iguais porém possuem capitalização diferente, por exemplo “escola” e “Escola”. Portanto, a função *clean_text* foi criada para cuidar disso:

```
def clean_text(msg):
    msg = msg.lower()
    msg = re.sub('[^a-zA-Z]', ' ', msg)
    msg = ' '.join(msg.split())
    return msg
```

Figura 11 - Código da função *clean_text*

Fonte: Elaborado pelos autores

A função *lower* transforma todas as letras para sua forma minúscula, e, em seguida, a função *sub*, da biblioteca *re*, transforma todos os caracteres não alfabéticos em espaços. Finalmente, a mensagem é fragmentada em uma lista de palavras utilizando a função *split* e depois são unidas de volta formando uma nova mensagem. O que essa linha de código faz é a remoção de espaços desnecessários, ou seja, garante-se que exista exatamente um espaço entre as palavras da mensagem.

Em seguida aplica-se um método de normalização que, como visto anteriormente, pode ser a *stemming* ou a *lemmatization*. Foi construída uma função para cada um dos métodos, a fim de experimentar diversos cenários e avaliar qual fornece o melhor resultado. Abaixo seguem os códigos de cada uma:

```
def stemming(msg):
    msg = clean_text(msg)
    stemmer = SnowballStemmer(language='english')
    wordlist = [stemmer.stem(word) for word in msg.split() if word not in
        set(stopwords.words('english'))]
    msg = ' '.join(wordlist)
    return msg
```

Figura 12 - Código da função *stemming*

Fonte: Elaborado pelos autores

```
def lemmatization(msg):
    msg = clean_text(msg)
    doc = nlp(msg)
    tokenlist = [token.lemma_ for token in doc if token not in nlp.Defaults.stop_words]
    msg = ' '.join(tokenlist)
    return msg
```

Figura 13 - Código da função *lemmatization*

Fonte: Elaborado pelos autores

É importante ressaltar que, dentro das funções apresentadas acima, a função *clean_text* é chamada, visto que é sempre desejável a sua utilização e também para que não seja necessário chamá-la no código.

Para a *stemming*, foi utilizado o *SnowballStemmer*, disponível na biblioteca NLTK. Esta função foi atribuída à variável *stemmer*. A mensagem, depois de passar pela função *clean_text*, é dividida em uma lista onde cada elemento é formado pelo radical (*stem*) de uma das palavras que compõem a mensagem. São também descartadas as palavras que estão na lista de *stopwords* na NLTK. Depois disso, as palavras são novamente reunidas em uma mensagem usando o método *join*, sendo separadas por um espaço.

Para a *lemmatization*, como comentado anteriormente, foi utilizada a biblioteca *SpaCy*. Primeiramente, como feito para a *stemming*, a função *clean_text* é aplicada à mensagem. A função *nlp* é uma variável que carrega a informação da função *load* (essa atribuição foi feita fora da função, e será mostrada mais à frente), que retorna uma série de informações referentes à mensagem, como falado anteriormente. Essas informações ficam armazenadas na variável *doc*. Novamente, é realizado um processo muito similar ao da função *stemming*, mas agora extrai-se o *lemma* de cada palavra da mensagem. Da mesma forma, descartam-se todas as palavras que sejam *stopwords*.

Com essas funções de pré-processamento definidas, é possível montar uma lista de mensagens pré-processadas, denominada *corpus*:

```
norm = lemmatization  # stemming, lemmatization
corpus = [norm(msg) for msg in list(dataset['msgContent'])]
```

Figura 14 - Código da criação do *corpus*

Fonte: Elaborado pelos autores

A variável *norm* pode ser tanto *lemmatization* como *stemming*, dependendo de qual técnica a ser utilizada. No código acima, esta normalização foi aplicada a cada linha da coluna *msgContent* do *dataset*, ou seja, a cada mensagem do *dataset*.

Além das funções de *stemming* e *lemmatization*, foi construída uma função que, dada uma lista de textos, retorna quais são suas palavras mais frequentes. Isso é interessante pois pode-se identificar palavras indesejadas que não estão na lista padrão de *stopwords*, que podem então ser inseridas para serem automaticamente descartadas e assim não interferirem no aprendizado do modelo.

```
def word_freq(wordlist, n):
    big_sen = ''

    for sen in wordlist:
        big_sen += sen

    return Counter(big_sen.split()).most_common(n)
```

Figura 15 - Código da função *word_freq*

Fonte: Elaborado pelos autores

A lista de textos foi denominada *wordlist* , sendo *n* o número de palavras a serem retornadas. As mensagens da lista são unidas em uma única mensagem, denominada *big_sen*, e a função *split* transforma a mensagem em uma lista com as palavras que a compõe. Então, a função *Counter* juntamente com a função *most_common*, ambas da biblioteca *collections*, retornam uma lista apenas com as “n” palavras mais frequentes.

Esta função foi utilizada fornecendo como parâmetros o *corpus* obtido e definindo *n* = 100; entre as palavras retornadas pela função, foram identificadas algumas indesejáveis para nosso modelo, sendo elas: "a", "e", "n", "s", "m", "o", "br", "ls", "www", "com", "fot", "cs", "hayasa" e "apollo". Portanto, estas palavras foram adicionadas às *stopwords* tanto da *SpaCy* como da NLTK:

```
nlp.Defaults.stop_words |= {'a', 'e', 'n', 's', 'm', 'o', 'br', 'ls', 'www', 'com',
                           'fot', 'cs', 'hayasa', 'apollo',}

for word in nlp.Defaults.stop_words:
    lex = nlp.vocab[word]
    lex.is_stop = True

stopwords.words('english').extend(['a', 'e', 'n', 's', 'm', 'o', 'br', 'ls', 'www', 'com',
                                   'fot', 'cs', 'hayasa', 'apollo'])
```

Figura 16 - Código da adição de novas *stopwords*

Fonte: Elaborado pelos autores

O mesmo procedimento acima foi repetido considerando a nova lista de *stopwords*, mas dessa vez não foi identificada nenhuma palavra indesejável.

Assim, com o *corpus* completamente tratado, pode-se prosseguir para a fase de criação da estrutura *Bag of Words*.

3.3.4 Criação da estrutura de dados Bag of Words

Para a criação dessa estrutura, foi utilizada a função *CountVectorizer,* disponível na biblioteca *sklearn*:

```
cv = CountVectorizer(max_features=max_ft)
X = cv.fit_transform(corpus)
```

Figura 17 - Código da criação das variáveis *cv* e *X*

Fonte: Elaborado pelos autores

A função foi atribuída à variável *cv*. A partir deste método pode-se utilizar a função *fit_transform* que, dada uma lista de frases, cria uma matriz onde as frases dessa lista são representadas nas linhas, e onde cada palavra existente nesse conjunto de frases é representada nas colunas. É importante notar que as palavras não são duplicadas, ou seja, existe apenas uma coluna para cada palavra, mesmo que ela apareça várias vezes na lista. Utiliza-se então esta função no *corpus* e a matriz gerada é atribuída à variável *X*, cujas dimensões são de 14.178 linhas e 6.734 colunas. O parâmetro *max_features* será comentado futuramente, na seção “Parâmetros do modelo”.

Em seguida, atribui-se à variável *y* a matriz de *targets* do *dataset*, ou seja, todas as colunas exceto a *msgContent*. Para isto, utilizamos a propriedade *iloc* da variável *dataset*:

```
y = labels_dataframe.values
```

Figura 18 - Código da criação da variável *y*

Fonte: Elaborado pelos autores

Assim, com a matriz de *features* (*X*) e de *targets* (*y*) feitas, pode-se avançar para a próxima etapa.

3.3.5 Mineração de Textos (aprendizado do modelo de classificação)

Esta é a principal parte do código. É nela onde é construído o classificador a ser utilizado, que é a função responsável pela geração do modelo. Segue o código abaixo:

```
method = BinaryRelevance  # BinaryRelevance, LabelPowerset
classifier = LogisticRegression  # LogisticRegression, SVC, GaussianNB

kf = KFold(n_splits=k)

for train_index, test_index in kf.split(X):
    X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[train_index], \
        y[test_index]

    cf = method(classifier(max_iter=max_iterations, random_state=42))
    cf.fit(X_train, y_train)
    y_pred = cf.predict(X_test)
```

Figura 19 - Código da criação e treinamento do classificador

Fonte: Elaborado pelos autores

Como visto anteriormente, escolheu-se como técnicas potenciais para a geração do modelo classificador a *LogisticRegression*, *SVC e GaussianNB*, e para os métodos de transformação de problemas, *BinaryRelevance* e *LabelPowerset*.

O aprendizado do modelo aplicou uma abordagem de validação cruzada (*cross validation*), muito comum em metodologias de descoberta de conhecimento. Na validação cruzada, estabelece-se um número (k) de “folds”, e o conjunto dos dados de entrada para o aprendizado é particionado em k subconjuntos. A partir daí, são

executadas k interações da técnica de mineração, onde em cada interação é aprendido um modelo utilizando-se 1 dos subconjuntos de dados para teste e os demais para treino do modelo. Ao final das k interações, os resultados dos k modelos construídos é agregado e calcula-se a média e o desvio padrão. A aplicação da abordagem de validação cruzada busca gerar modelos mais robustos a variações nos dados de entrada, e evitar que o modelo construído seja super ajustado para reproduzir as regras específicas de apenas um subconjuntos dos dados de entrada (um fenômeno conhecido como *overfitting*).

Para exemplificar, suponha que k = 5. Então o tamanho do conjunto de teste será de ⅕ (ou 20%) do dataset, e do conjunto de treino será de 80%. Serão gerados 5 modelos diferentes, em cada um variando quais 20% do *dataset* estamos atribuindo ao conjunto de teste (o conjunto de treino é sempre o resto, que também varia). Por fim, avalia-se cada modelo gerado baseando-se nas métricas e guarda-se o resultado em suas respectivas listas.

A construção do modelo dentro de cada iteração do loop da validação cruzada aconteceu da seguinte forma. A variável *kf* foi criada para armazenar a função *KFold*, da metodologia de validação cruzada descrita. Os valores de *k* serão discutidos na seção de parâmetros do modelo. Com o número *k* definido, utilizamos a função *split* para dividir o *dataset*. Essa função retorna, para cada iteração, os índices de uma fração do dataset de tamanho 1/*k*, atribuídos à variável *test_index*, e os índices da parte restante do dataset, de tamanho 1 - (1/k), atribuídos à variável *train_index*. Dessa maneira, a cada iteração gera-se um modelo diferente, pois os conjuntos de treinamento e de teste mudam em função da partição das matrizes *X* e *y*, dividindo-as em conjuntos de treinamento e conjuntos de teste. O conjunto de treinamento é o par *X_train* e *y_train*, enquanto o de teste é o par *X_test* e *y_test*. Em seguida, foi atribuída à variável *cf* a informação de um método de transformação de problemas aplicado a um classificador, sendo ambos parâmetros a serem escolhidos. Assim como para *max_features*, discutiremos os valores de *k* e de *max_iterations* na seção “Parâmetros do modelo”. O *random state*, quando especificado, habilita a replicabilidade do modelo, ou seja, ao usar o mesmo número os resultados serão os mesmos. O número 42 foi escolhido por ser um valor comumente utilizado. Por fim, ainda em cada iteração foi utilizado o método *fit* da variável *cv* que realiza o ajuste do modelo baseado nos dados de treinamento. Com isso, a variável *cv* guarda a informação do modelo

ajustado de cada iteração. Finalmente, com o modelo ajustado às informações dos dados do conjunto de treinamento, pode-se avaliar como ele se desempenha com novos dados, isto é, os dados do conjunto de teste. Depois de realizar-se a predição das respostas do *X_test*, guardado na variável *y_pred*, compara-se estas respostas com as respostas reais *y_test*, com o objetivo de avaliar o desempenho do modelo de cada iteração.

A seguir revela-se uma parte maior do código, que engloba a parte mostrada na seção anterior:

```
acc_array = []
hamm_loss_array = []

kf = KFold(n_splits=k)

for train_index, test_index in kf.split(X):
    X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[train_index], \
        y[test_index]

    cf = method(classifier(max_iter=max_iterations, random_state=42))
    cf.fit(X_train, y_train)
    y_pred = cf.predict(X_test)

    acc_array.append(accuracy_score(y_test, y_pred))
    hamm_loss_array.append(hamming_loss(y_test, y_pred))

acc = np.mean(acc_array)
hamm_loss = np.mean(hamm_loss_array)
```

Figura 20 - Código da criação e treinamento do classificador (completo)

Fonte: Elaborado pelos autores

Escolheu-se realizar a avaliação do modelo através de duas métricas:

- *Accuracy* (acurácia): número de mensagens classificadas corretamente sobre o número total de mensagens. No caso *multi-label*, uma mensagem classificada corretamente significa que todos os rótulos estão certos (SCIKIT-LEARN, 2017). Se temos, por exemplo, um conjunto de predições ([0, 1], [0, 0]) e a resposta verdadeira é ([0, 0], [0, 0]), temos uma *accuracy* de ½ = 0,5 ou 50%.
- *Hamming loss*: número de classificações erradas sobre o número total de classificações (SCIKIT-LEARN, 2017). Tomando como exemplo os mesmos conjuntos acima, temos um *hamming loss* de ¼ = 0,25 ou 25%.

Após o loop da validação cruzada, os resultados de avaliação de cada modelo são guardados em listas específicas para cada métrica, cujos nomes são *acc_array* e *hamm_loss_array*, utilizando a função *append*. Aqui, é importante notar que estas listas foram inicializadas antes do *loop*, como mostrado no código.

Sendo assim, com os resultados de cada modelo, realiza-se uma média de cada uma utilizando a função *mean*, da biblioteca *numpy*. Assim, obtém-se os resultados médios para as duas métricas, armazenados nas variáveis *acc* e *hamming_loss*.

Após o aprendizado dos modelos dentro do loop de validação cruzada, escolhe-se o modelo com melhor desempenho, o qual será então utilizado como classificador para então classificar automaticamente novos dados.

3.4 Metodologia experimental para construção do modelo

A metodologia experimental seguida neste trabalho para o aprendizado do modelo de classificação cobriu diversos cenários, configurados variando-se os valores do conjunto de parâmetros das técnicas utilizadas. Alguns destes parâmetros já foram citados nas seções anteriores, mas esta seção tem como objetivo explicitar cada um deles:

- Norm (*normalization*) {*stemming*, *lemmatization*}: estratégia de normalização de dados adotada.
- SW (*stopwords*) {*Yes*, *No*}: com este parâmetro, pode-se incluir ou não as *stopwords* adicionais que foram definidas ao utilizar a função *word_freq*.
- Method {BR (*BinaryRelevance*), LP (*LabelPowerset*)}: método de transformação de problemas
- Max_ft (*max_features*) {1000, 2000, 4000, 6000}: este é um parâmetro da função *CountVectorizer*, tendo como finalidade limitar o número de palavras incluídas no modelo, para que se descartem palavras irrelevantes. O critério de escolha é sempre incluir das palavras mais frequentes para as menos frequentes.

- CF (*classifier*) {GNB (*GaussianNB*), LR (*LogisticRegression*), SVC}: determina qual classificador será utilizado para criar e ajustar o modelo.
- Max_iter (*max_iterations*) {40, 50, 60, 80, 100, 300, 500, 700}: esse é um parâmetro da função do classificador, que determina qual o número máximo de iterações são permitidas. Importante notar que este parâmetro não está presente no classificador *GaussianNB*.
- k {5, 10}: determina em quantas partições o dataset será dividido, e, consequentemente, quantos modelos diferentes serão criados na validação cruzada.

Vale notar aqui que as opções de escolha dos parâmetros numéricos foram limitadas devido à limitações computacionais e de tempo. As combinações de valores incluídas no experimento cobrem, no entanto, um conjunto razoável e abrangente para construção de um modelo de classificação com bom potencial para generalizar os resultados obtidos no experimento para classificar novos conjuntos de dados. Na seção seguinte, descreve-se como esses parâmetros foram combinados para definir diferentes cenários do experimento realizado.

3.5 Experimento inicial

Antes de realizar todas as combinações de parâmetros planejadas, foram realizados testes iniciais para determinar e fixar alguns parâmetros, pois uma combinação de todas as opções de todos os parâmetros geraria uma quantidade muito grande de testes.

Os primeiros parâmetros escolhidos foram *normalization* e *stopwords*. Portanto, para determinar seus valores ideais, os demais parâmetros foram mantidos fixos. Na tabela 2 são vistos os parâmetros utilizados e resultados dos 4 primeiros testes:

Tabela 5 - Parâmetros e resultados dos 4 primeiros testes

#	NORM	SW	METHOD	MAX_FT	CF	MAX_ITER	k	ACC	HAMM_LOSS
1	Lemma	Yes	BR	4000	LR	50	5	79,04%	1,18%
2	Lemma	No	BR	4000	LR	50	5	79,04%	1,18%
3	Stem	Yes	BR	4000	LR	50	5	77,02%	1,27%
4	Stem	No	BR	4000	LR	50	5	77,02%	1,27%

Fonte: Elaborado pelos autores

Percebe-se, pelos resultados das métricas, que a *lemmatization* é superior à *stemming*, e que as *stopwords* adicionais não impactam no resultado, provavelmente por seu um conjunto pequeno de palavras. Assim, com os resultados deste experimento inicial, decidiu-se continuar apenas com a *lemmatization* e seguir com as *stopwords* adicionais incluídas.

Como terceiro e último parâmetro a ser fixado e determinado, escolheu-se o *classifier*. Na tabela abaixo seguem os testes realizados, incluindo-se também o teste #1 por ter sido o melhor, e também por já conter o classificador LogisticRegression:

Tabela 6 - Parâmetros e resultados do teste 1 e dos testes de 5 a 10

#	NORM	SW	METHOD	MAX_FT	CF	MAX_ITER	k	ACC	HAMM_LOSS
1	Lemma	Yes	BR	4000	LR	50	5	79,04%	1,18%
5	Lemma	Yes	BR	4000	GNB	-	5	4,42%	19,37%
6	Lemma	Yes	BR	4000	SVC	50	5	4,33%	13,83%
7	Lemma	Yes	BR	4000	SVC	100	5	11,82%	9,53%
8	Lemma	Yes	BR	4000	SVC	300	5	31,58%	4,91%
9	Lemma	Yes	BR	4000	SVC	500	5	59,26%	3,12%
10	Lemma	Yes	BR	4000	SVC	700	5	64,68%	2,79%

Fonte: Elaborado pelos autores

Percebe-se imediatamente que o *GaussianNB* não é um bom classificador para esse tipo de problema, apresentando uma acurácia muito baixa e hamming loss alto. Foram realizados vários testes com o SVC tentando variar o *max_iter*, mas mesmo para valores altos, ficou bem longe do desempenho obtido ao usar o *LogisticRegression*. Por esse motivo, decidiu-se seguir com o *LogisticRegression* como o classificador.

3.6 Experimento

A partir de agora, deseja-se realizar todas as combinações dos parâmetros definidos previamente, para determinar quais são os seus melhores valores. Entretanto, para os testes iniciais, o código foi executado alterando-se os parâmetros um a um, manualmente, o que daria um enorme trabalho para fazer com o resto dos testes. Por conta disso, um novo código foi criado, praticamente igual ao primeiro, mas com algumas diferenças que o tornasse capaz de gerar esses testes automaticamente.

Primeiramente, foram importadas todas as bibliotecas a serem usadas no código, e incluiu-se as funções *clean_text* e *lemmatization*:

```
import pandas as pd
import re
import spacy
import numpy as np
import os

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import KFold
from skmultilearn.problem_transform import BinaryRelevance, LabelPowerset
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import hamming_loss, accuracy_score

def clean_text(msg):
    msg = re.sub('[^a-zA-Z]', ' ', msg)
    msg.lower()
    return msg

def lemmatization(msg):
    msg = clean_text(msg)
    doc = nlp(msg)
    tokenlist = [token.lemma_ for token in doc if token not in nlp.Defaults.stop_words]
    msg = ' '.join(tokenlist)
    return msg
```

Figura 21 - Código dos *imports* e funções a serem usadas na automatização do código

Fonte: Elaborado pelos autores

A ideia para a automatização do experimento foi criar uma tabela, no Excel, contendo os valores dos parâmetros de cada teste a ser realizado. Abaixo seguem as 5 primeiras linhas desta tabela:

Tabela 7 - Conteúdo das 5 primeiras linhas da tabela de parâmetros

NORM	SW	METHOD	MAX_FT	CF	MAX_ITER	k
lemmatization	Yes	BinaryRelevance	1000	LogisticRegression	40	5
lemmatization	Yes	BinaryRelevance	1000	LogisticRegression	60	5
lemmatization	Yes	BinaryRelevance	1000	LogisticRegression	80	5
lemmatization	Yes	BinaryRelevance	1000	LogisticRegression	100	5
lemmatization	Yes	BinaryRelevance	2000	LogisticRegression	40	5

Fonte: Elaborado pelos autores

Então, este arquivo foi salvo no formato CSV com o nome “Parametros.csv” e depois carregado no código, sendo as informações dessa tabela atribuídas a uma variável denominada *df_param* utilizando-se a função *read_csv* da bilbioteca *pandas*:

```
script_path = os.path.dirname(__file__)

df_param = pd.read_csv(os.path.join(script_path, 'Parametros.csv'), delimiter = ';')
```

Figura 22 - Código do carregamento do arquivo contendo os parâmetros

Fonte: Elaborado pelos autores

Em seguida, assim como no código anterior, o *dataset*, *labels_dataframe* e o *corpus* foram criados, com a única diferença de que a normalização está fixa como sendo o *lemmatization*:

```
dataset = pd.read_excel(os.path.join(script_path, 'Datasets\Dataset OneHot.xlsx'))
labels_dataframe = dataset.iloc[:, 1:]

corpus = [lemmatization(msg) for msg in list(dataset['msgContent'])]
```

Figura 23 - Código da criação das variáveis *dataset, labels_dataframe* e corpus para o código automatizado

Fonte: Elaborado pelos autores

Depois disso, vem a maior parte do código, onde são feitas iterações sobre cada um dos testes registrados na planilha de parâmetros:

```
for i in indexes:
    method = eval(df_param.iloc[i,2])
    max_ft = df_param.iloc[i,3]
    classifier = eval(df_param.iloc[i,4])
    max_iterations = df_param.iloc[i,5]
    k = df_param.iloc[i,6]

    cv = CountVectorizer(max_features=max_ft)
    X = cv.fit_transform(corpus)

    acc_array = []
    hamm_loss_array = []

    kf = KFold(n_splits=k)

    for train_index, test_index in kf.split(X):
        X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[train_index], \
            y[test_index]

        cf = method(classifier(max_iter=max_iterations, random_state=42))
        cf.fit(X_train, y_train)
        y_pred = cf.predict(X_test)

        acc_array.append(accuracy_score(y_test, y_pred))
        hamm_loss_array.append(hamming_loss(y_test, y_pred))

    acc = np.mean(acc_array)
    hamm_loss = np.mean(hamm_loss_array)

    m_acc_array.append(acc)
    m_hamm_loss_array.append(hamm_loss)
```

Figura 24 - Código do iterador do código automático

Fonte: Elaborado pelos autores

Primeiramente, como no código anterior. atribui-se à variável *y* a matriz de *targets*.

Logo após, através da propriedade *shape,* é obtida a quantidade de linhas existentes na tabela, armazenando-a na variável *lines*, que representa o número de testes a serem feitos. Repare que essa propriedade retorna dois valores, o número de linhas e de colunas, e, como só estamos interessados no número de linhas, colocamos a variável "_" para receber o valor do número de colunas. Na linguagem Python, este é um caracter especial que indica que esta variável não será armazenada na memória.

Em seguida, foi criada uma lista do *numpy* para guardar os índices de cada teste, guardando-a na variável *indexes*. Para gerar essa lista, foi utilizada uma compreensão de lista iterando sobre a função *range*, que teve como parâmetro o número de linhas.

Depois, é feita uma iteração por essa lista de índices, e a cada iteração as informações de cada coluna da tabela de parâmetros, na linha *i*, são atribuídas à sua respectiva variável, sendo elas: *method, max_ft, classifier, max_iterations* e *k*. O método *eval* utilizado transforma um texto/número (que é o formato do valor retornado pela função *iloc* aplicada na tabela de parâmetros) em uma função a ser avaliada. Por exemplo, o texto “BinaryRelevance” é transformado para a função BinaryRelevance.

Depois disso, os dados são estruturados utilizando-se o *Bag of Words*, como visto no código anterior.

Logo em seguida, destacado pelo fundo mais claro, vem um grande pedaço de código que é exatamente igual ao visto na seção 4.3.5. A ideia é aplicar aquele mesmo processo de treinamento e avaliação dos modelos, mas a cada iteração, a cada conjunto de parâmetros.

Depois de rodar esse loop, os valores médios das métricas para aquela iteração são armazenados nas variáveis *m_acc_array* e *m_hamm_loss_array*, que foram inicializadas como listas antes do *loop* principal.

Finalmente, ainda é preciso exportar os resultados obtidos de cada teste, e aqui foi escolhido exportar para uma planilha Excel, utilizando o seguinte código:

```
df_excel = pd.DataFrame({'Test':indexes+1,'Accuracy':m_acc_array,'Hamming Loss':
                         m_hamm_loss_array})

df_excel.to_excel(os.path.join(script_path, 'Resultados.xlsx'), index=False)
```

Figura 25 - Código da exportação dos resultados para Excel

Fonte: Elaborado pelos autores

A função *Dataframe*, da biblioteca *pandas*, cria uma tabela tendo como parâmetros os nomes das colunas e as informações de cada coluna. No caso, foi incluída uma coluna com o índice do teste, e uma coluna para cada métrica. Finalmente, a tabela foi exportada para Excel utilizando-se a função *to_excel*.

4. RESULTADOS

As informações com os resultados de cada teste foram consolidadas com as informações dos parâmetros em apenas uma planilha Excel. Com o algoritmo automático foram gerados 64 testes, além dos 10 testes iniciais.

Como foram escolhidas duas medidas de avaliação (*accuracy* e *hamming loss*), foi necessário criar uma terceira métrica que ponderasse estas duas, para que fosse possível ranquear as soluções obtidas. Para isso, dentro do Excel, foram criadas duas novas colunas, ACC_RANK e HAMM_RANK (uma para cada métrica), onde seus valores são uma normalização das suas respectivas métricas. A normalização foi feita seguindo a equação 3, representada abaixo.

$$x' = (x - min(x)) \div (max(x) - min(x)) \quad (3)$$

Aqui é importante clarificar que foram deixados de fora da normalização os testes iniciais que tiveram como classificador o SVC ou o *GaussianNB* e como normalização o *stemming*. Isso foi feito pois esses testes foram consideravelmente piores, o que iria polarizar a normalização, visto que os valores a serem comparados ficariam muito próximos uns dos outros.

Em seguida, foi criada uma terceira coluna, denominada FINAL SCORE, cujos valores são a média aritmética das duas colunas criadas anteriormente, sendo esses os valores da terceira métrica. Esses valores vão de 0 a 1, sendo 1 o melhor possível.

Abaixo, seguem duas tabelas, contendo informações das 10 melhores soluções (ranqueadas de maior para menor segundo a coluna FINAL SCORE). A tabela com todos os testes encontra-se em anexo.

Tabela 8 - Parâmetros das 10 melhores soluções

#	NORM	SW	METHOD	MAX_FT	CF	MAX_ITER	k
69	Lemma	Yes	LP	4000	LR	80	10
65	Lemma	Yes	LP	2000	LR	80	10
70	Lemma	Yes	LP	4000	LR	100	10
66	Lemma	Yes	LP	2000	LR	100	10
73	Lemma	Yes	LP	6000	LR	80	10
72	Lemma	Yes	LP	6000	LR	60	10
74	Lemma	Yes	LP	6000	LR	100	10
68	Lemma	Yes	LP	4000	LR	60	10
64	Lemma	Yes	LP	2000	LR	60	10
61	Lemma	Yes	LP	1000	LR	80	10

Fonte: Elaborado pelos autores

Tabela 9 - Resultados das 10 melhores soluções

#	ACC	HAMM_LOSS	ACC RANK	HAMM RANK	FINAL SCORE
69	83,35%	1,17%	0,997	0,736	0,867
65	83,35%	1,17%	0,997	0,733	0,865
70	83,37%	1,18%	1,000	0,724	0,862
66	83,31%	1,18%	0,988	0,711	0,849
73	83,30%	1,18%	0,985	0,713	0,849
72	83,32%	1,19%	0,989	0,692	0,841
74	83,28%	1,19%	0,982	0,697	0,839
68	83,27%	1,19%	0,980	0,681	0,831
64	83,19%	1,19%	0,962	0,673	0,818
61	83,01%	1,21%	0,923	0,640	0,782

Fonte: Elaborado pelos autores

Observa-se que as soluções com o método *LabelPowerset* prevaleceram nas posições mais altas. Isso se deu pois, apesar de o método *BinaryRelevance* ter desempenhado ligeiramente melhor na medida *hamming loss*, o *LabelPowerset* teve um desempenho bem superior na medida *accuracy*.

Outro parâmetro que se manteve constante nas melhores soluções foi o *k*, que, entre os valores 5 e 10 testados, o valor “10” entregou resultados bem melhores. Considerando esse valor de *k*, podemos afirmar que as soluções são bem robustas, visto que os resultados dessas métricas são a média de 10 modelos diferentes. Os parâmetros *max_iter* e *max_ft* não tiveram muito impacto nos resultados.

Sendo assim, foi obtida uma solução, a de número 69, que teve como resultados uma *accuracy* de 83,35% e um *hamming loss* de 1,17%. Isso quer dizer que o modelo conseguiu predizer as categorias corretamente em 83,35% das

entradas do conjunto de teste, e que apenas 1,17% das classificações foram erradas (o modelo disse que era “1” mas era “0”, ou vice-versa, lembrando-se que o problema é multi-label). Esses resultados foram bem satisfatórios, considerando que este é um problema de NLP multi-label, com 24 categorias diferentes.

5. CONCLUSÃO

Este trabalho aplicou uma metodologia de Data Science, dentro da qual foi criado um algoritmo em Python para classificação multi-label de sentenças em linguagem natural em categorias correspondentes a atos de fala específicos, através da utilização de técnicas de Processamento de Linguagem Natural e Mineração de Textos. Com base nos objetivos propostos inicialmente para este trabalho, foi possível atingir todos e também obter resultados satisfatórios. Além de terem sido revistas as teorias de KiP e de Atos de Fala e os conceitos das principais técnicas de NLP, o algoritmo criado foi aplicado para classificar as mensagens entre participantes de um processo intensivo em conhecimento real em 24 atos de fala. O algoritmo criado conseguiu atingir uma *accuracy* de 83,35% e um *hamming loss* de 1,17%, o que mostra que o modelo obtido é adequado para ser implementado no dia a dia de uma empresa real. Entretanto, é importante relembrar que este algoritmo é adaptado apenas para dados escritos na língua inglesa. Além disso, os resultados encontrados se baseiam no dataset que foi passado para ser usado como base para este estudo, que contém um gabarito para a classificação dos Atos de Fala de cada mensagem. Logo, se no futuro houver alguma mudança nesse gabarito, o trabalho precisará passar por uma revisão.

Com a criação desse algoritmo, espera-se contribuir para que os gestores do processo de gestão de incidentes abordado no estudo de caso consigam identificar padrões de comunicação que frequentemente levam a um resultado positivo ou negativo. Em outras palavras, será possível pensar em possíveis melhorias a partir da construção de um painel de indicadores de desempenho, que levará em conta métricas relacionadas a comunicação entre os participantes do KiP.

Neste trabalho foi implementada uma técnica bem simples de estruturação dos dados, o *Bag of Words*. Para trabalhos futuros, seria interessante aplicar uma técnica mais sofisticada, como por exemplo *TF-IDF*, *word2vec* e *GloVe Embeddings*.

Além disso, há também a possibilidade de serem executados mais testes na base de dados, em especial testes out-sample, isto é, aplicados ao conjunto de dados que não tenha sido considerado no aprendizado do modelo.

REFERÊNCIAS BIBLIOGRÁFICAS

Richetti, Pedro Henrique Piccoli, João Carlos de AR Gonçalves, Fernanda Araujo Baião, and Flávia Maria Santoro. "Analysis of knowledge-intensive processes focused on the communication perspective." In International Conference on Business Process Management, pp. 269-285. Springer, Cham, 2017.

GONÇALVES, João Carlos. COGNITIVEKiP – A COGNITIVE BPM THEORY FOR KNOWLEDGE-INTENSIVE PROCESSES, 2018.

AUSTIN, J.L.: How to do things with words. Oxford university press (1962)

MARCONDES, Danilo. Speech Act Theory as a pragmatic view on language. 2006. Tese (Filosofia) - UNISINOS, [*S. l.*], 2006.

Como NLP e chatbot podem trabalhar juntos?. [*S. l.*], 17 abr. 2019. Disponível em: https://www.zenvia.com/blog/nlp-e-chatbot. Acesso em: 17 maio 2020.

OLIVEIRA, Wallace. Passo a passo para a gestão de incidentes ITIL. [*S. l.*], 14 maio 2017. Disponível em: https://www.venki.com.br/blog/gestao-de-incidentes-itil/. Acesso em: 27 maio 2020.

BROWNLEE, Jason. What Is Natural Language Processing?. [*S. l.*], 22 set. 2017. Disponível em: https://machinelearningmastery.com/natural-language-processing/. Acesso em: 6 jun. 2020.

JACKOV, Luchezar. Proceedings of Recent Advances in Natural Language Processing: Feature-Rich Part-Of-Speech Tagging Using Deep Syntactic and Semantic Analysis. Institute for Bulgarian Language Bulgarian Academy of Sciences, Hissar, Bulgaria, p. 224-231, 7 set. 2015.

KADAM, Shirish. Dependency Parsing in NLP. [*S. l.*], 31 mar. 2019. Disponível em: https://medium.com/@5hirish/dependency-parsing-in-nlp-d7ade014186. Acesso em: 6 jun. 2020.

COSTA, C. D. Python Libraries for Natural Language Processing: An Overview Of popular python libraries for Natural Language Processing. [*S. l.*], 28 abr. 2020. Disponível em: https://towardsdatascience.com/python-libraries-for-natural-language-processing-be0e5a35dd64. Acesso em: 6 jun. 2020.

GITHUB. FAQ: Answers to Frequently Asked Questions about NLTK. [*S. l.*], 5 jul. 2019. Disponível em: https://github.com/nltk/nltk/wiki/FAQ. Acesso em: 6 jun. 2020.

HEIDENREICH, Hunter. Stemming? Lemmatization? What?: Taking a high-level dive into what stemming and lemmatization do for natural language processing tasks and how they do it.. [*S. l.*], 21 dez. 2018. Disponível em: https://towardsdatascience.com/stemming-lemmatization-what-ba782b7c0bd8. Acesso em: 6 jun. 2020.

PORTER, Martin. An algorithm for suffix stripping. [*S. l.*], 2001. Disponível em: https://tartarus.org/martin/PorterStemmer/def.txt. Acesso em: 6 jun. 2020.

PORTER, Martin. The English (Porter2) stemming algorithm. [*S. l.*], 2001. Disponível em: http://snowball.tartarus.org/algorithms/english/stemmer.html. Acesso em: 6 jun. 2020.

PAICE, Chris D. Another Stemmer. ACM SIGIR Forum, Department of Computing Lancaster University, p. 56-61, 1 nov. 1990.

SNOWBALL. Stemming algorithms. [*S. l.*], 2001. Disponível em: https://snowballstem.org/algorithms/. Acesso em: 6 jun. 2020.

LEONEL, Jorge. Classification Methods in Machine Learning. [*S. l.*], 9 out. 2019. Disponível em: https://medium.com/@jorgesleonel/classification-methods-in-

machine-learning-58ce63173db8#:~:text=Classification%20is%20a%20supervised%20machine,learning%20to%20classify%20new%20observations. Acesso em: 6 jun. 2020.

BROWNLEE, Jason. 4 Types of Classification Tasks in Machine Learning. [*S. l.*], 8 abr. 2020. Disponível em: https://machinelearningmastery.com/types-of-classification-in-machine-learning/20supervised%20machine,learning%20to%20classify%20new%20observations. Acesso em: 6 jun. 2020.

SCHULTEBRAUCKS, Lasse. Gaussian Naive Bayes. [*S. l.*], 23 ago. 2017. Disponível em: https://medium.com/@LSchultebraucks/gaussian-naive-bayes-19156306079b. Acesso em: 6 jun. 2020.

SCIKIT-LEARN. 1.9. Naive Bayes. [*S. l.*], 2019. Disponível em: https://scikit-learn.org/stable/modules/naive_bayes.html. Acesso em: 6 jun. 2020.

WANG, Xinge. The Math behind Linear SVC Classifier. [*S. l.*], 2 ago. 2018. Disponível em: https://www.kaggle.com/xingewang/the-math-behind-linear-svc-classifier. Acesso em: 6 jun. 2020.

BROOKS, Keith. Day (10) — Machine Learning — Using LogisticRegression with scikit-learn. [*S. l.*], 26 mar. 2018. Disponível em:

BROWNLEE, Jason. Logistic Regression for Machine Learning. [*S. l.*], 1 abr. 2016. Disponível em: https://machinelearningmastery.com/logistic-regression-for-machine-learning/. Acesso em: 6 jun. 2020.

RAULJI , J. K.; SAINI, J. R. Stop-Word Removal Algorithm and its Implementation for Sanskrit Language. International Journal of Computer Applications, [*S. l.*], v. 150, n. 2, p. 15-17, 1 set. 2016.

LUACES, Oscar *et al*. Binary relevance efficacy for multilabel classification. Progress in Artificial Intelligence, [*S. l.*], v. 1, p. 303-313, 14 out. 2012.

SZYMANSKI, Piotr; KAJDANOWICZ, Tomasz. Scikit-multilearn: A scikit-based Python environment for performing multi-label classification. Journal of Machine Learning Research, [*S. l.*], p. 1-15, 5 fev. 2017.

SKLEARN.METRICS.ACCURACY_SCORE. [*S. l.*], 2017. Disponível em: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html. Acesso em: 7 jun. 2020.