André Teixeira
Baião Alves

Automação de sistemas fechados usando Inteligência Artificial

Closed Loop automation through Intelligence Artificial

Universidade de Aveiro
2022

André Teixeira
Baião Alves

Automação de sistemas fechados usando Inteligência Artificial

Closed Loop automation through Intelligence Artificial

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia em Engenharia Informática, realizada sob a orientação científica do Doutor Mário Luís Pinto Antunes, Professor adjunto convidado do ISCA do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Rui Luís Andrade Aguiar, Professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor José Nuno Panelas Nunes Lau
Professor Associado, Universidade de Aveiro

vogais / examiners committee

Prof. Doutor João Nuno Vinagre Marques da Silva
Professor Auxiliar Convidado, Universidade do Porto - Faculdade de Ciências - Departamento de Ciência de Computadores

Prof. Doutor Rui Luís Andrade Aguiar
Professor Catedrático, Universidade de Aveiro

agradecimentos / acknowledgements

Ao longo desta caminhada, houve pessoas que ajudaram-me na realização desta dissertação e apoiaram-me desde do início desta jornada.

Em primeiro lugar gostava de agradecer sobretudo aos meus pais, que embora estejamos longe fisicamente na maioria dos meses, tentavam sempre manter contacto comigo e davam-me força para continuar.

Gostaria de agradecer aos meus amigos que, como vivenciámos as mesmas dificuldades, criámos uma ligação forte e ajudavam-nos uns aos outros para que todos chegassem ao fim.

Por fim, agradecer aos meus orientadores por estarem sempre presentes, pelas suas ideias e conhecimento, que ajudaram na realização desta tese.

E, um obrigado a todas as pessoas que ajudaram de alguma forma durante este processo.

Palavras Chave

CNN, diagnósticos de alarmes, *FastText*, LSTM, *machine learning*, modelos de *word embedding*,rede neural, relações de semelhança, vocabulário, *Word2Vec*

Resumo

Atualmente, há uma necessidade de automação de processos aplicados nas redes devido à elevada complexidade e tamanho das mesmas. Nas redes das operadoras de telecomunicações registam diariamente eventos de alarmes que ocorreram nos seus dispositivos. Estes equipamentos como são de fornecedores ou operadoras diferentes, geram diagnósticos de falhas que utilizam nomenclaturas distintas para se referirem à mesma causa da falha. Deste modo, neste trabalho desenvolveu-se um modelo que mede relações de semelhança entre os termos que aparecem nos diagnósticos de falhas, na medida de tornar possível mapear os alarmes para um modelo único alarmístico. Inicialmente, processou-se uma base de dados de diagnósticos de falhas reais com intuito de treinar modelos de *word embedding*, tais como, *Word2Vec* e *FastText*, para converter as palavras em vetores numéricos. Portanto, para avaliar os modelos, gerou-se uma base de dados a partir de um *captcha* de palavras. Este foi utilizado por especialistas da área com objetivo de encontrarem pares de termos semelhantes. Através das suas respostas foi possível medir as suas respetivas similaridades, sendo consideradas como as esperadas. Contudo, os modelos de *word embedding* demonstraram não ter capacidade de encontrar este tipo de relações. Por isso, adicionou-se uma camada de modelos de machine learning, nos quais recebiam os vetores dos pares definidos na base de dados e tinham que prever a similaridade mais próxima da esperada. Com isto, uma rede neural simples com os vetores de 128 dimensões gerados pelo modelo *Word2Vec* com uma arquitetura CBOW obteve os melhores resultados, com valores de 0.95 e 0.90 de coeficientes de correlação de *Pearson* e *Spearman*, respetivamente. A CNN com vetores da mesma dimensão, mas com uma arquitetura *skip-gram* no Word2Vec obteve apenas 0.22 de correlação de Pearson e 0.23 de Spearman. As features geradas combinadas com a LSTM obteve-se valores de correlação próximos de zero, exceto com os vetores de 384 de dimensão gerados pelo *Word2Vec* com uma arquitetura CBOW, que conseguiram obter 0.62 como coeficiente de correlação de Pearson e 0.55 de Spearman. A CNN e LSTM embora sejam redes muito mais complexas, a base de dados não tem tamanho suficiente para este tipo de redes conseguirem encontrar uma boa função que meça a similaridade entre as palavras do vocabulário específico de redes e software.

Keywords Alarm diagnostics, CNN, FastText, LSTM, Machine Learning, Neural Network, Similarity relations, Vocabulary, Word Embedding Models, Word2Vec

Abstract

Nowadays, given the networks complexity and size there is a need for process automation especially malfunction correction. Every day there are many failures in the devices, which, as they are from different vendors or belong to distinct telecommunications operators, alarm diagnostics use different vocabularies to refer to the exact cause of the failure. Thus, in this work, a model was developed that finds relations of similarity between these terms so that it is possible to map the alarms to a single alarmist model. Initially, a database of real fault diagnostics was processed to train embedding word models, such as Word2Vec and FastText, to convert the words into numeric vectors. Therefore, to evaluate the models, it is necessary to have a minimal amount of data, hence the creation of a captcha system to collect pairs of similar terms and measure the similarity between new acquired terms. However, word embedding models are not capable to find this type of relationships. Therefore, a layer of machine learning models was added, in which they received the vectors of the pairs defined in the database and had to predict the closest to the expected similarity. With this, the simple neural network has achieved the best results, while CNN and LSTM although they are much more complex network the database is not large enough to achieve good results. Thus, a neural network with 128-dimensional vectors generated by the Word2Vec model with a CBOW architecture achieved the best results, with final values of 0.95 and 0.90 of Pearson and Spearman correlation coefficients, respectively. The CNN with vectors of the same dimension buy with a skip-architecture in Word2Vec had only 0.23 Pearson basis and 0.23 Spearman basis. The features combined with the LSTM achieved low results values, except for the 384-dimensional vectors generated by Word2Vec with a CBOW architecture, with values of 0.62 of Pearson's correlation coefficient and 0.55 of Spearman's.

Conteúdo

Lista de Figuras

Lista de Tabelas

Glossário

3GPP	3rd Generation Partnership Project
5G	5th Generation Mobile Network
BoW	Bag of Words
BERT	Bidirectional Encoder Representations from Transformers
biLM	Bidirectional Language Model
CBOW	Continuous Bag-of-Words
CNN	Convolutional Neural Network
DT	Decision Tree
DP	Distributional Profiles
ELMo	Embedding from language Models
GRU	Gated Recurrent Unit
GloVe	Global Vectors
IDF	Inverse Document Frequency
LR	Logistic Regression
LSTM	Long short-term memory
IP	Internet Protocol
HS	Hierarchical Softmax
ML	machine learning
MAE	Mean Absolute Error
MSE	Mean Squared Error
NV	Naïve Bayes
NLTK	Natural Language Toolkit
NS	Negative Sampling
NSP	next sentence prediction
NP	noun-phrases
PoS	Part of Speech
PMI	Point Mutual Information
PLN	Processamento de Linguagem Natural
RF	Random Forest
RNN	Recurrent Neural Networks
ReLU	Rectified Linear Units
RMSprop	Root Mean Square Propogation
RMSE	Root Mean Squared Error
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
TF	Term Frequency
TF-IDF	Term Frequency - Inverse Document Frequency
UE	User Equipment

CAPÍTULO 1

Introdução

Nos últimos anos, é notável o rápido desenvolvimento de tecnologias de comunicação e um aumento de serviços integrados na vida das pessoas, o que tende a aumentar o número de utilizadores. Com o surgimento do 5G, as redes de telecomunicações têm vindo continuamente a expandir, o tipo de dispositivos tem aumentado e tecnologias de comunicação têm sido atualizadas de forma a corresponder às necessidades dos clientes. Isto faz com que seja cada vez mais difícil gerir as redes devido à falta de processos de automação, aumento de custos e complexidade de reparação de falhas [1].

Efetivamente, há uma necessidade de automação de processos para que uma rede consiga ser um sistema *self-healing*, ou seja, a rede consiga detetar, diagnosticar e reparar falhas com o mínimo de intervenção humana. As operadores de telecomunicações têm uma infraestrutura de rede normalmente composta por dispositivos como switches, routers, firewalls, hubs, fontes de alimentação e entre outros, que são equipamentos de vários fornecedores, nos quais suportam os seus serviços. Isto faz com que a disponibilidade da rede seja muito importante para as operadoras, porque caso haja interrupções na rede, pode ser prejudicial para o negócio. Portanto, os diagnósticos de falhas nos dispositivos têm que ser eficientes, para que um especialista da área possa localizar e reparar a falha o mais rápido possível. Cada dispositivo envia informação sobre o seu estado e desempenho, no entanto, quando há uma falha, é lançado um evento de alarme e normalmente é propagado ao longo da rede para múltiplos dispositivos que estão conectados entre si, o que cria uma grande quantidade de dados de alarmes [2].

1.1 Motivação

Uma rede celular fornece conetividade sem fio para dispositivos que estão em movimento, como por exemplo, telemóveis, tablets, carros e entre outros, sendo estes mais conhecidos por User Equipment (UE). O núcleo de uma rede de 5G, que foi definida pelo 3rd Generation Partnership Project (3GPP), utiliza uma arquitetura baseada em serviços, alinhada com uma

solução nativa de *cloud*, que como se pode ver na figura 1.1, o *Mobile Core* abrange todas as funções e interações do 5G [3].

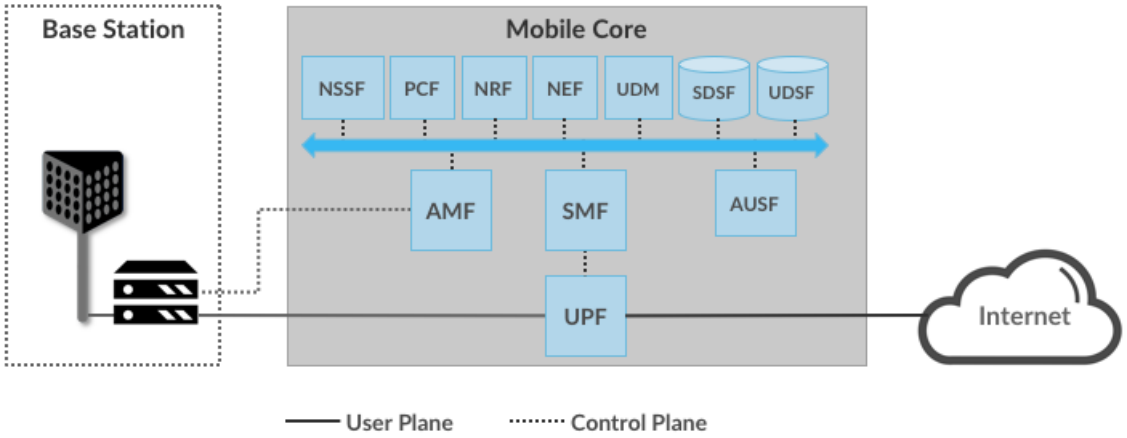


Figura 1.1: Arquitetura base de uma rede 5G. [Fonte: [4]]

O 5G tem um papel importante num futuro de dispositivos interconectados, porque consegue conexões ultrarrápidas e com pouca latência. Isto porque o 5G, ao contrário do 4G, envia e recebe dados no espetro de rádio em frequências mais altas para atingir velocidades maiores. O problema destas frequências maiores é que sofrem muito quando encontram obstáculos no seu caminho, como por exemplo, edifícios [5]e até pessoas. Portanto, para impulsionar este tipo de redes é necessário um grande número de dispositivos para cobrir uma certa área. Para além disso, tendo em conta que acontecem diariamente centenas de alarmes e a complexidade das redes, muitas operadoras e especialistas do domínio concordam que a deteção de falhas e o procedimento de diagnóstico manual é muito lento e não viável tecnicamente e financeiramente [1]. Portanto, como entre dispositivos de diferentes fornecedores e de operadoras de telecomunicações existem múltiplos formatos de alarmes, se houvesse uma representação uniforme dos diagnósticos de falhas dos dispositivos, poderia facilitar os mecanismos autónomos de resolução de erros, como por exemplo, *root cause analysis*, correlações e resolução.

1.2 Objetivos

A informação apresentada nos alarmes são cadeias de caracteres minimamente organizados, porém o vocabulário é distinto dependendo da marca ou fornecedor do dispositivo. Por isso, este trabalho tem como objetivo desenvolver um modelo que consiga interpretar as nomenclaturas utilizadas pelos diferentes fabricantes, de modo a encontrar sinónimos entre os termos de domínio específico de redes e *software*. Isto, com finalidade de mapear os diversos diagnósticos de falhas dos dispositivos de diversos fabricantes para um modelo único de alarmes, ou seja, se consiga identificar que diferentes relatórios de alarmes representam a mesma falha.

O modelo proposto implementa técnicas de Processamento de Linguagem Natural (PLN) para extrair informação das palavras e de *machine learning* (ML) de modo a encontrar relações

de sinonímia entre as palavras. Será utilizado uma base de dados onde contém registos de diagnósticos de alarmes provenientes da rede de telecomunicações da Nokia. Esta base de dados é privada e, por isso, será anonimizada para não divulgar informação interna da empresa.

1.3 Estrutura da dissertação

Para facilitar a leitura do leitor, a estrutura da dissertação consiste primeiramente no capítulo 2, que apresenta o estado de arte, onde é abordado vários conceitos e modelos relacionados com a área de PLN. De seguida, o capítulo 3 que descreve a solução proposta para a resolução do problema desta dissertação e como foi implementada. No capítulo 4 apresenta os resultados obtidos através das experiências realizadas. Logo após, tem-se o capítulo 5 onde se expõem as conclusões e uma análise final baseada no capítulo anterior. Por fim, o último capítulo é relativo às referências.

CAPÍTULO 2

Estado de Arte

Neste capítulo é descrito vários conceitos e trabalhos que suportarão as decisões que se poderá enfrentar durante o desenvolvimento do modelo proposto. Inicialmente, é importante perceber o que é PLN, como se pode aplicar, tipo de *features* que se pode extrair, modelos de ML e, por fim, trabalhos relacionados.

2.1 Processamento de Linguaguem Natural

PLN é uma área que estuda técnicas e métodos para construir modelos que sejam capazes de analisar linguagem natural para desempenhar tarefas como reconhecimento da fala, análise de sentimentos e entre outros [6]. As primeiras aplicações do PLN eram baseadas em regras, que consistem num grande volume de código para definir as várias condições. No entanto, ML tem sido muito adotado em vários trabalhos pela sua capacidade de fornecer simples soluções para problemas complexos através da análise de uma grande quantidade de dados em pouco tempo. Uma parte das tarefas de PLN são aplicadas em texto não estruturado, o que é necessário inicialmente ser realizado um pré-processamento do texto para que depois se consiga extrair *features* de forma a alimentar os modelos de ML. Portanto, as próximas secções abordam esses vários passos.

2.2 Pré-processamento do texto

Antes de qualquer tarefa de PLN, é importante haver um pré-processamento do texto que será analisado. As bases de dados textuais podem conter palavras indesejáveis, pontuação, números e entre outros. Por isso, é importante remover este tipo de ruído e manter um formato consistente nos dados para que depois se utilize modelos capazes de analisar e aprender consoante o problema a que se é aplicado. Pode haver tarefas específicas que este processo pode diferir, como por exemplo, processar documentos científicos que contêm equações matemáticas será diferente do que processar texto relacionado com comentários de utilizadores nas redes sociais.

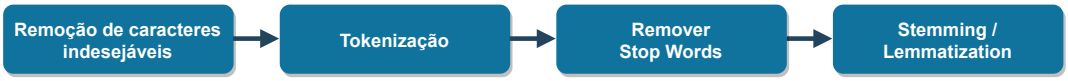


Figura 2.1: *Pipeline* típica de pré-processamento de uma tarefa PLN.

Portanto, este tópico irá-se focar nas técnicas mais utilizadas na área de PLN, para compreender como funcionam e a sua importância.

Na figura 2.1 pode-se visualizar uma *pipeline* típica de pré-processamento, que começa por **remover e substituir todos os caracteres indesejáveis**. O próximo passo típico é a **tokenização**, no qual é um processo que transforma o texto em vários segmentos, que são conhecidos como tokens. Se pensarmos num documento constituído por várias frases, este processo o que faz é partir essas frases em várias palavras. Deste modo, simplifica o processo da análise das mesmas.

De seguida, realiza-se a **remoção de *Stop-words***, no qual é um processo muito utilizado em problemas relacionados com texto, para remover todo o tipo de palavras insignificantes que não adicionam informação ao contexto da frase e que aparecem com muita frequência. Por exemplo, palavras como "a", "the", "is", "and" e entre outras. Para isso, existem bibliotecas, tais como, *NLTK* e *Scikit-learn*, que já disponibilizam uma lista de *stop-words*.

As palavras têm múltiplas formas de serem escritas, mas têm o mesmo significado. Portanto, é importante aplicar um processo que reduza as palavras na sua forma raiz como o método ***Stemming***. Por exemplo, as palavras "reading" e "readed" podem ser reduzidas na forma "read". Para isso, existem vários algoritmos, tais como, *Porter Stemmer* e *Snowball Stemmers*. O algoritmo *Snowball* é uma melhoria do *Porter*, sendo que é mais agressivo ao cortar as palavras conseguindo obter um melhor desempenho.

Para além disso, existe o método ***Lemmatization*** que tem o mesmo objetivo que o *stemming*, só que no caso do *stemming* corta o início ou o fim da palavra consoante uma lista de prefixos e sufixos comuns, enquanto que *lemmatization* tem em conta a morfologia da palavra. Embora o *stemming* seja muito mais fácil para processar as palavras, o *lemmatization* tem uma melhor exatidão, porque tem uma compreensão profunda linguisticamente para formar o glossário que permite ao algoritmo procurar a parte significativa da palavra [7]. O que torna o *lemmatization* preferível quando é importante a análise contextual.

Por fim, existem outros métodos que podem ser utilizados dependendo da tarefa de PLN, que extraem informação das palavras. No âmbito de encontrar relações entre palavras, ***Part of Speech* (PoS)** pode ser um método muito relevante, é uma técnica que associa uma tag a cada token para explicar como a palavra é usada na frase, ou seja, não tem haver com a semântica das palavras em si, mas com a sintaxe das mesmas. Por exemplo, se o token for um nome, associa uma tag "NN", caso seja um verbo associa uma tag "VB", independentemente do significado do nome ou verbo. PoS é uma solução de aprendizagem supervisionada [6] muito utilizada para encontrar padrões no texto, com o intuito de criar relações entre as palavras ou de gerar *features* para que os modelos de ML consigam ter mais informação para treinar e possam obter uma melhor performance.

2.2.1 Deteção e substituição de abreviações

A deteção e substituição de abreviações é uma técnica para encontrar e estender acrónimos e siglas na sua forma longa, como por exemplo, transformar a sigla SO para Sistema Operativo. Este tipo de processo é principalmente um desafio na área da medicina, porque existe uma grande quantidade de registos eletrónicos relacionados com a saúde dos pacientes, onde é utilizado abreviações com muita frequência [8]. No entanto, as técnicas podem ser traduzidas para outro tipo de cenários de domínio específico, que podem ser importantes inserir na *pipeline* de pré-processamento.

Uma forma para resolver este problema, poderia ser extrair abreviações de documentos relacionados com um domínio de modo a ter uma cobertura de todas as as abreviações para depois ser feito um mapeamento. Contudo, seria custoso computacionalmente para construir e manter atualizado. No entanto, existem métodos automáticos que conseguem detetar abreviações e identificar o seu significado baseados em ML. Para detetar abreviações, no estudo [9] comparara os seguintes classificadores de ML: *Decision Tree* (DT), *Support Vector Machine* (SVM) e *Random Forest* (RF). Para treinar os modelos utilizaram relatórios clínicos de resumos de altas dos pacientes, nos quais era feita a identificação manual das abreviações através de profissionais da saúde. SVM foi o modelo mais robusto que conseguiu obter um F-score de 94.5%. Após ser feita a deteção de abreviações, é necessário descobrir o seu significado, no qual, na área da medicina podem existir vários significados para a mesma abreviação e esta desambiguação é um passo importante. O estudo [10] usou vários corpus onde extraíu-se pares de abreviações e sua respetiva longa forma através de padrões textuais. Com isto, os autores utilizaram vários classificadores de ML, tais como, *Naïve Bayes* (NV), *Logistic Regression* (LR), SVM e *Monte-Carlo Sampling Logistic Regression*, que eram treinados através de instâncias positivas, ou seja, os pares retirados diretamente do texto e de instâncias negativas, que eram geradas a partir da mistura entre pares das suas abreviações e definições. Deste modo, conseguiram obter bons resultados, atingindo um F-score de 86.20%, 93.52% de precisão e 84.64% de recall.

Tendo em conta que na área da saúde existem uma grande quantidade de abreviações comparativamente à área da informática e redes, os modelos de ML de aprendizagem supervisionada podem funcionar muito bem. Contudo, para obter bons resultados é necessário ter um bom corpus para conseguir formar dados para treinar os modelos.

2.2.2 Deteção e tratamento de *noun-phrases* (NP)

Há palavras compostas que remetem para o mesmo objeto que devem ser consideradas como um token e não em vários. Por exemplo, a expressão "command line" deve ser vista como um token, senão perde o sentido principal. Este tipo de tokens são nomeados como NP, nos quais a sua deteção e o seu tratamento são passos que podem ser importantes serem aplicados numa *pipeline* de pré-processamento para que os NP sejam analisados de uma forma mais correta durante a resolução do problema.

Existem várias abordagens para detetar os NP baseados em algoritmos de ML, sendo modelos com uma aprendizagem supervisionada ou não.

Em relação aos algoritmos de ML de aprendizagem supervisionada, exigem que haja instâncias categorizadas a identificar se são NP ou não para treinar os modelos de classificação. Para isso, requer que haja pessoas capazes de anotar NP de forma a criar um conjunto de dados de treino. No caso de cenários de domínio específico, as pessoas precisam de ter conhecimento do domínio para desempenhar esta tarefa [11].

No caso de aprendizagem não supervisionada, há muitos trabalhos, tal como, [12] e [13] que para detetar NP utilizam um modelo estatístico que usa a fórmula *Point Mutual Information* (PMI), no qual é uma medida da sobreposição de informações entre duas variáveis aleatórias. Quanto mais as duas variáveis ocorrerem em conjunto num dado corpus, maior será o valor do PMI e, caso passe um determinado valor de *threshold*, estas palavras são consideradas NP. Tendo em conta o exemplo anterior, é um caso de um *bigram*, no entanto existem casos de *trigrams* e *quadrigrams*, nos quais podem ser mais difíceis de detetar devido à reduzida probabilidade de as palavras ocorrerem em conjunto, comparativamente à probabilidade das mesmas ocorrerem separadamente. Após a deteção, o tratamento de NP pode ser feito através da concatenação dos n-grams com o símbolo `"_"` e passa a ser apenas um token [13]. Por outro lado, tendo em conta que as palavras podem ser representadas por vetores, que será explicado na secção 2.3.4, outro tratamento que pode ser feito é a média entre os valores dos vetores das diferentes palavras [14]. Em ambos os métodos de tratamento, como não é feita uma avaliação direta, é difícil definir qual é o melhor.

Por outro lado, esta técnica poderia-se relacionar com a técnica de transformar as abreviações na sua forma longa, pois quando se estende uma abreviação obtêm-se vários tokens referentes ao mesmo objeto, o que portanto a forma longa de uma abreviação poderia-se definir de imediato como um NP.

2.3 Relações entre as palavras

As relações entre as palavras têm uma aplicação importante no PLN. Nesta secção irá-se focar principalmente em relações de sinonímia e hiperonímia. Duas palavras são consideradas sinónimas quando na mesma frase podem ser trocadas entre elas e o significado do contexto não muda, este tipo de relação tem um papel essencial em sistemas de recuperação de informação através da expansão de *queries*. No caso das relações de hiperonímia, é quando o campo semântico de um termo cobre o sentido do outro termo e esta relação entre hiperónimo-hipónimo é muito conhecida como a relação de *is-a*, como por exemplo, *red is a color*. Em PLN, a relação hiperónimo-hipónimo é fundamental em tarefas como *named entity recognition* e *question-answering*, como questões do tipo *what is*. Portanto, existem vários tipos de técnicas para extrair as relações referidas de forma automática, que seguem essencialmente as seguintes abordagens: *pattern-based* e *ML-based*. Para além disso, irá-se analisar redes semânticas que são estruturas, nas quais representam ligações entre termos de um determinado vocabulário. Isto para perceber como as redes foram construídas e como se pode aplicar de modo a ajustar relações definidas. Por fim, será abordado vários métodos que têm como objetivo representar palavras em forma numérica, sendo um processo muito importante, porque é uma forma de

extrair *features* a partir do texto, que servem como base para os modelos de ML estabelecerem relações entre as palavras.

2.3.1 *Pattern-based*

O uso de padrões textuais é uma das primeiras abordagens utilizadas para extrair relações semânticas entre as palavras. Na literatura é muito citado os padrões de Hearst [15], que consiste em uma listagem de padrões léxico-sintáticos para extrair hiperónimos e hipónimos aprendidos de forma manual ou semi-automática, como por exemplo, "X such as P". Por outro lado, também existem padrões para encontrar sinónimos [16], contudo este tipo de método normalmente apresenta um valor de *recall* baixo por não incluir todo o tipo de padrões. Por isso, apareceram outros estudos para aumentar este *recall*, como no caso de [17] que utiliza PoS para tagar as palavras de modo a cobrir mais padrões textuais, em vez de procurar apenas sequências de palavras. Mais recentemente, investigadores [18] extraíram mais de 400 milhões de relações de hiperonímia do corpus web *Common Crawl* através de padrões textuais léxico-sintáticos. Mesmo assim, este tipo de método apresenta várias desvantagens devido ao facto de ser ineficiente, porque está sujeito a ter muito ruído por natureza e é difícil generalizar para cenários de domínio específico.

2.3.2 *ML-based*

Na literatura de PLN, muitas tarefas optam por algoritmos de *machine learning* de forma a alcançar os resultados de estado de arte. Muitos investigadores aplicam modelos ML com aprendizagem não supervisionada, onde aplicam técnicas de *clustering* e aprendizagem supervisionada de forma a induzir automaticamente regras através dos dados de treino [19].

Aprendizagem Não Supervisionada

Aprendizagem não supervisionada usa algoritmos de ML para analisar e agrupar dados que não estão rotulados. Estes algoritmos encontram padrões escondidos e agrupam dados sem intervenção humana. Tendo em conta um espaço vetorial, como os modelos de *word embedding* projetam as palavras no espaço, é importante analisar vários algoritmos que se possa aplicar, que sejam capazes, por exemplo, de dividir as palavras em diferentes grupos consoante a similaridade entre elas.

O K-means é um dos algoritmos de *clustering* mais conhecido que consiste em inicializar vários pontos no espaço, sendo que cada um é um *cluster*, e para cada ponto é associado a um *cluster* que esteja mais próximo. Por fim, recalcula-se os centros dos *clusters*. K-means consegue obter boas performances quando os dados estão espacialmente separados e utiliza a distância *Euclidean* como métrica, ou seja, a distância entre dois pontos é medida através do comprimento do segmento de linha entre os mesmos.

Spectral Clustering é outro algoritmo que refere-se à *spectral decomposition* da matriz de similaridade aplicada aos dados de forma a reduzir a dimensão dos mesmos antes de realizar o *clustering*. Cada vetor é calculado a similaridade com todos os outros vetores, resultando assim a matriz de similaridade. *Spectral Clustering* é um algoritmo que tem a capacidade de agrupar os dados de modo a convergir para uma solução ótima global [20].

Um outro algoritmo que apresenta boa performance em agrupar palavras com a mesma semântica é o *Brown Clustering*, este aplica *hierarchical agglomerative clustering* das palavras baseado nos contextos em que elas ocorrem, deste modo maximiza a qualidade do modelo de linguagem, nomeadamente *cluster n-gram model* onde as probabilidades das palavras são baseadas nos *clusters* de palavras anteriores. Embora seja um algoritmo que tenha incorporado um modelo de linguagem, o *Spectral Clustering* consegue obter resultados semelhantes [21], o que se torna um modelo de *clustering* útil para várias tarefas de PLN, devido ao facto de, como utiliza uma representação de baixa dimensões, poderá se adicionar novas features de modo a remover palavras *outliers* de *clusters* ou criar relações entre *clusters*.

Aprendizagem Supervisionada

Aprendizagem supervisionada é das abordagens mais populares em PLN que utiliza dados de treino com *labels*, que pode ser dividido em dois tipos de abordagem: regressão para quando os valores de saída do modelo é um valor real e classificação em que o modelo associa os valores de entrada a uma categoria ou classe. Portanto, na aprendizagem supervisionada, para cada exemplo de dados de treino já se sabe previamente os valores que o modelo deve prever ou a que categoria a que pertencem. À medida que os dados de treino são inseridos no modelo, este ajusta os seus pesos até que o modelo fique adequadamente ajustado de forma a conseguir fazer boas previsões a partir de novos dados desconhecidos, tendo uma boa exatidão.

Começando por modelos mais simples, existem muitos classificadores que são utilizados nas tarefas de PLN, o que se irá focar apenas nos mais conhecidos ou utilizados que tendem apresentar melhores resultados. Um dos modelos é o *Support Vector Machine* (SVM), que é um modelo que tem apresentado bons resultados em vários problemas de diferentes campos, como por exemplo, categorização de texto, por isso tem sido muito explorado em problemas relacionados com PLN. A sua principal ideia é encontrar um *hyperplan*, que é representado como um vetor, de forma a conseguir separar duas categorias para que novos exemplos ao serem mapeados no espaço, se consiga definir a qual categoria pertencem. O *Naïve Bayes* (NV) é um classificador probabilístico muito simples e muito utilizado em vários campos como o SVM, baseado no teorema de *Bayes*. Outro modelo muito utilizado em problemas de classificação, é o *Decision Tree* (DT) devido ao facto de apresentar bons resultados em diferentes problemas, como por exemplo, deteção de spam,sendo um algoritmo fácil de compreender [22]. Efetivamente, o conceito consiste em uma árvore de decisão, em que cada nó pode ser visto como um teste, os ramos como resultados e os nós folhas representam as diferentes categorias que um exemplo pode pertencer.

Há muitos modelos que usam redes neurais como base que funcionam muito bem em tarefas de PLN. Mas, primeiro é importante perceber o que é uma rede neural. Rede neural [23] é um algoritmo de inteligência artificial que tenta encontrar relações no conjunto de dados através de processos semelhantes ao cérebro humano. A rede é composta por camadas de nós, contendo uma camada de entrada que recebe os dados, uma ou mais camadas escondidas e, por fim, uma camada de saída. Cada nó tem associado pesos e limites. Sempre que o valor de saída de um nó individual for acima do limite, este nó é ativado e envia os dados para a

próxima camada de nós. Caso contrário, nenhum dado é passado para a próxima camada. Os valores de saída são calculados a partir de funções de ativação que recebem a soma dos pesos e um *bias*. Existem vários tipos de funções de ativação, como por exemplo, Rectified Linear Units (ReLU) é uma função que se o valor é negativo a função retorna zero, caso seja um número positivo, retorna diretamente o valor. *Sigmoid* é outra função, conhecida como função logística, que utiliza a fórmula 2.1, na qual o resultado retorna um valor entre zero e um.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

A equipa Google Brain propuseram uma função de ativação, nomeadamente Swish, que consiste na fórmula 2.2, onde demonstraram em várias experiências e em base de dados desafiadoras que a função conseguia obter melhores resultados em modelos mais profundos comparativamente à função de ativação ReLU, por ser uma função mais suave e não monótona [24].

$$f(x) = x * sigmoid(x) \tag{2.2}$$

É preciso perceber também como as redes atualizam os pesos dos seus nós. A cada iteração realizada na fase de treino dos modelos, é atualizado os seus parâmetros internos de modo a que se obtenha o menor erro possível calculado pela função de custo. Este função quantifica o erro a partir dos valores previstos e esperados. Ao longo do treino, tem-se como objetivo encontrar os parâmetros que se obtenha o mínimo custo possível. Na escolha dos parâmetros, existem funções de otimização que adaptam os parâmetros de formas diferentes. Uma das mais conhecidas é o SGD [25], que em cada iteração atualiza os parâmetros com apenas uma instância dos dados de treino e mantém o *learning rate*, isto é, o tamanho do passo com que o algoritmo tenta se aproximar ao mínimo de custo a cada iteração. O otimizador Root Mean Square Propogation (RMSprop) é um método que ao contrário do SGD não tem um *learning rate* fixo, ou seja, o *learning rate* é adaptativo consoante a média da magnitude dos gradientes nos pesos, como por exemplo, depende da rapidez com que os gradientes estão a mudar. Isto faz com que seja mais difícil ultrapassar o mínimo de custo. Outra otimizador muito popular é o Adam [26], um método estocástico de gradiente descendente que se baseia na estimativa adaptativa de momentos de primeira ordem como o caso do RMSprop, no entanto, também se adapta aos momentos de segunda ordem, ou seja, adapta o *learning rate* tendo em conta não apenas a média da magnitude dos gradientes, como também a variância não centrada do mesmo.

Após se perceber como uma rede funciona durante o seu treino, existem outros modelos que usam redes mais profundas e complexas, que é importante serem analisadas pelas suas capacidades de resolução dos problemas. Um das redes é a RNN que tem a capacidade de lidar com longas sequências de token como entrada, o que é muito utilizado em vários problemas PLN [27], tais como, compreensão de linguagem falada e tradução automática. A arquitetura do modelo RNN geralmente contém três camadas: *input layer*, *hidden layer* e *output layer*, sendo uma rede *feedforward* em que utiliza o *feedback* dos nós anteriores como entrada dos

próximos nós. Tendo em conta o diagrama da figura 2.2, à esquerda é possível visualizar a arquitetura do modelo RNN e no lado direito pode-se ver que é possível desenrolar em uma rede mais completa, dependendo do tamanho da frase, ou seja, se a sequência tem três tokens, a rede terá que desdobrar a rede em 3-*layer neural network*, uma para cada palavra.

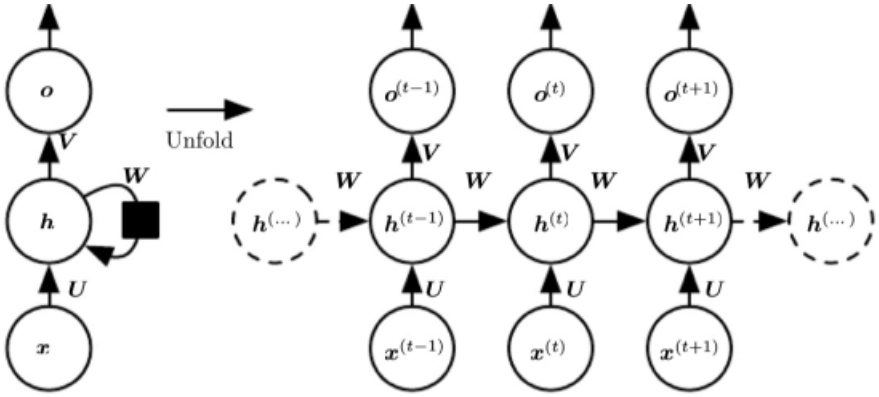


Figura 2.2: Diagrama do modelo RNN.[Fonte: *Yuan Ma*]

No entanto, durante o *back propagation*, o modelo RNN sofre com o problema *vanishing gradient*. O gradiente são valores usados para atualizar os pesos da rede e este diminui à medida que se propaga de volta ao longo do tempo, o que torna a aprendizagem difícil. Isto resulta em uma distorção nos resultados do modelo, o que faz com que não consiga prever corretamente [28]. Por isso, realizaram-se pequenas mudanças no modelo, surgindo o modelo *Long short-term memory* (LSTM) de modo a eliminar este problema.

LSTM é uma variante do modelo RNN que tem a capacidade de aprender dependências de longo prazo e tem mecanismos internos chamados *gates* que regulam o fluxo de informação. A ideia principal dos modelos LSTM está centrada no estado da célula e dos seus *gates*, como se pode observar na figura 2.3. O estado da célula transfere a informação por toda a cadeia de sequência, o que pode carregar informações relevantes ao longo do processamento da sequência. Isto faz com que informações de etapas iniciais consigam chegar a etapas posteriores, reduzindo o efeito de short-term memory. As *gates* permitem decidir se a informação pode ser adicionada ou não à célula de estado.

Outra variante de RNN é o *Gated Recurrent Unit* (GRU), que é muito semelhante ao LSTM, única diferença é que em vez de ter o estado de célula para transferir informação, tem o *hidden state* e tem apenas *duas gates*, a *reset gate* e *update gate*. A *update gate* tem a mesma função que a forget e input gate do modelo LSTM, ou seja, decide se deve descartar ou adicionar informação. *Reset Gate* é usada para decidir quanta informação passada deve ser esquecida. GRU consegue ser mais simples e mais rápido do que LSTM, por isso alguns investigadores optam por esta arquitetura.

As várias pesquisas realizadas para encontrar relações entre as palavras, normalmente alimentam as diferentes arquiteturas de RNN com sequências de palavras ou tokens que são transformados em vetores através dos modelos *word embedding*. Para além disso, existem investigações que utilizam sequências de PoS tags [27], que acreditam que possa ser mais

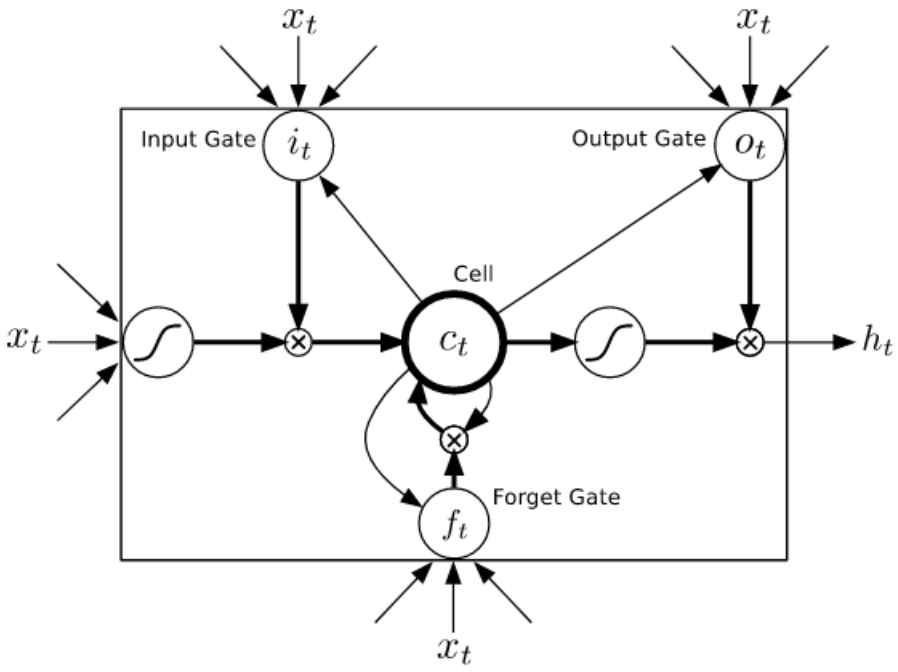


Figura 2.3: Célula LSTM.[Fonte: [29]

vantajoso do que utilizar *word embeddings*.

Por fim, uma rede neural profunda muito efetiva que é aplicada em tarefas de PLN é a *Convolutional Neural Network* (CNN), na qual é constituída por camadas Convolucionais e por camadas de *Pooling* [30]. A camada de convolução consiste em filtros, conhecidos também por *kernel*, que consiste numa matriz de $m \times n$ dimensões que irá deslizar ao longo da matriz de entrada verticalmente e horizontalmente. Como se pode ver na figura 2.4, é calculado o produto escalar entre o *input* e o filtro, como no caso da sub-matriz destacada a azul, após calcular o produto escalar com os pesos da matriz do filtro obtém-se o valor 4 que será guardado na matriz de saída.

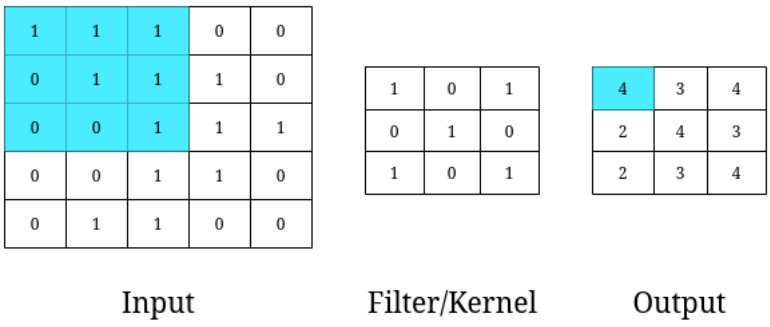


Figura 2.4: Exemplo de como o filtro é aplicado numa camada convolucional. Neste caso o filtro de tamanho 3×3 irá percorrer a matriz de entrada 5×5 com um deslize igual a 1.

A camada de *Pooling* tem como objetivo abstrair as *features* provenientes da camada convolucional, funciona da mesma forma que a mesma, só que em vez de calcular o produto escalar entre sub-matrizes dos dados de entrada e o *kernel*, é aplicado uma função agregadora. A função é o *max* que tem a responsabilidade de identificar as *features* mais importantes

ou relevantes numa sequência de valores. Portanto, o processo consiste em uma janela de tamanho $n \times m$ percorrer a matriz de entrada e guardar o valor mais alto na matriz de saída.

A CNN é muito usada para processar imagens que alimentam a rede com matrizes que contêm os valores dos pixeis. Para ser aplicado em tarefas de PLN, pode-se pensar que é preciso extrair *features* das palavras. Portanto, através de *embedding* das palavras é possível construir matrizes em que cada linha representa o vetor de uma palavra. Deste modo, a rede consegue resolver tarefas desta área.

Os modelos de ML após terem sido treinados, há várias métricas para perceber a qualidade dos modelos. Assim sendo, irá-se focar nas principais métricas utilizadas em aprendizagem supervisionada, onde muitos trabalhos da área de PLN usam para compararem entre si os seus resultados.

Num problema de classificação binária, a partir dos dados de teste, os modelos irão prever a que categoria deveriam pertencer. Com isto, é possível ter *True Positives*(TP), *True Negatives*(TN), *False Positives*(FP) e *False Negatives*(FN), que respetivamente correspondem quando o modelo associa o rótulo corretamente como positivo, negativo e associa incorretamente o rótulo como positivo e negativo. A partir destes valores, podem ser calculadas várias métricas, tais como:

- Precisão mede a percentagem de TP tendo em conta todos os positivos que foram previstos.

$$P = \frac{TP}{TP + FP}$$

- *Recall* mede a percentagem de TP tendo em conta todas as instâncias que são realmente positivas.

$$R = \frac{TP}{TP + FN}$$

- F1-score é definido como uma média harmónica entre a precisão e *recall*.

$$F = \frac{2 * precisão * recall}{precisão + recall}$$

- Exatidão calcula o número de previsões que o modelo fez corretamente tendo em conta todas as previsões que fez.

$$A = \frac{TP + TN}{TP + TN + FP + FN}$$

Num problema de regressão, não se pode calcular as métricas anteriores, porque os valores de saída são números reais, portanto o que se faz é calcular o erro entre os valores atuais r e os valores previstos p. As métricas mais usadas neste tipo de problemas são:

- Mean Squared Error (MSE) calcula o quanto os valores previstos se desviaram dos valores corretos, sendo uma boa métrica para comparar entre modelos de regressão.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (r_i - p_i)^2$$

- Root Mean Squared Error (RMSE) é basicamente a raíz quadrada de MSE. Esta métrica é mais comum ser utilizada, porque o erro é mais pequeno, sendo mais prático para comparar com resultados de outros modelos. Por outro lado, o MSE é calculado o erro ao quadrado e, portanto, a raiz quadrada traz de volta ao mesmo nível de erro de previsão e facilita a sua interpretação.
- Mean Absolute Error (MAE) é calculado de forma semelhante ao MSE, só que em vez de somar o erro ao quadrado, soma o valor absoluto do erro. Desta forma, MAE trata os erros todos da mesma forma, enquanto que MSE como é ao quadrado, dá uma maior penalização a erros maiores.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|r_i - p_i|$$

Estas métricas podem ser úteis para avaliar os modelos de ML de modo a ser possível discutir qual é o que tem uma melhor performance no contexto do problema desta dissertação.

2.3.3 Redes Semânticas

As redes semânticas são redes que representam conhecimento em estrutura de grafo, no qual cada nó são entidades e as suas ligações representam as suas relações com outros nós. Estas relações são aprendidas estatisticamente e linguisticamente através de fontes de conhecimento, como por exemplo, Wikipédia. Este tipo de redes é aplicável em várias áreas, tais como, mineração de dados de texto, PLN, recuperação de informação e entre outras. Para medir a semelhança entre palavras, existem métodos baseados em dicionários ou ontologias que se pode calcular através da distância em que duas palavras se encontram numa estrutura hierárquica ou adição de uma classificação. Por outro lado, podem servir para expansão de uma *query* ou vocabulário. Portanto, é importante conhecer algumas redes que existem, que podem ajudar a definir relações de semelhanças entre as palavras e como podem ser construídas.

Há redes semânticas *open-source* públicas que podem ser utilizadas: uma das redes semânticas mais conhecidas é o *WordNet* [31], uma base de dados lexical em inglês de grande escala que foi construída manualmente por especialistas, onde contém nomes, verbos, adjetivos e advérbios em conjuntos de sinónimos, em que cada um é representado como um conceito. Entre cada conjunto, existem vários tipos de relações, tais como, sinónimos, antónimos, hiperónimo, hipónimo e entre outros. Mais tarde apareceu *ConceptNet* [32] que representa conhecimento também numa estrutura organizada em grafo que foi construído com uma aprendizagem não supervisionada. Esta rede liga palavras e frases que foram extraídas de várias fontes, como por exemplo, Wikipédia, *WordNet* e *Wiktionary*, que formam relações, tais como, *PartOf*, *UsedFor*, *IsA* e entre outras. No entanto, *WordNet* e *ConceptNet* utilizam fontes de senso comum, o que para relacionar palavras da área da engenharia não é muito útil, devido ao facto de o vocabulário não conter uma variedade de termos técnicos e tecnológicos. Com isto, surgiram outras redes, como no caso de *B-Link* [33], que os seus autores utilizaram a técnica de *web crawler* com o objetivo de percorrer páginas de vários sites da internet e extrair a informação mais relevante, o que por conseguinte foi possível processar um milhão de artigos

em diferentes campos da engenharia e vários blogs. Através destes recursos, construíram a rede semântica direcionada para engenharia usando uma aprendizagem não supervisionada, aplicando uma análise na rede que consiste nas métricas probabilidade e velocidade para correlacionar conceitos que foram extraídos. Em 2020, surgiu a rede *TechNet* [34] que cobre um vocabulário de vários termos relacionados com vários domínios da tecnologia, extraídos de uma base de dados de patentes. Para tal, utilizou-se técnicas de PLN e de modelos de *word embedding*, tais como, *Word2Vec* e Global Vectors (GloVe) para vetorizar as palavras e definir relações entre elas, tendo em conta a distância em que os vetores se encontram no espaço vetorial.

Na tabela 2.1 pode-se ver as diferentes redes que foram referidas e é importante referir que dependendo da rede, os dados que se utiliza é algo a ter em conta para quando se vai construir uma rede semântica. Ao construir um vocabulário, se quer que seja focado num domínio específico, é essencial que se encontre base de dados ou fontes relacionadas com este domínio, para que se consiga representar o conhecimento da melhor forma.

	Redes Semânticas	**Tipo de abordagem**	**Fontes**	**Relações**
Vocabulário que não está relacionado com engenharia	Wordnet	Manualmente	Especialistas	Sinonímia, Antonímia, Hiponímia, Meronímia, Toponímia
	ConceptNet	Aprendizagem não supervisionada	Open Mind Common Sense, Wiktionary, Open Multilingual Wordnet, Wikipédia, JMDict, OpenCyc	Total de 34 tipos de relações. Por exemplo, *PartOf*, *UsedFor* e *IsA*
Vocabulário relacionado com engenharia	B-Link	Aprendizagem não supervisionada	Artigos, Blogs	Distância na rede normalizada
	TechNet	Aprendizagem não supervisionada	Patentes	Similaridade do cosseno

Tabela 2.1: Visão geral da redes semânticas mencionadas.

2.3.4 Extração de *features*

As *features* têm um papel importante na aprendizagem de modelos de ML. Contudo, este tipo de modelos não conseguem processar texto diretamente, por isso em PLN é necessário métodos para representar as palavras em forma numérica. Nesta subsecção irá-se abordar técnicas que transformam palavras em vetores, tais como, Bag of Words (BoW), TF-IDF, Distributional Profiles (DP) e modelos de *word embedding*.

BoW e TF-IDF

Inicialmente é importante salientar um dos modelos mais simples, nomeadamente o modelo BoW[35], que representa o texto em números, no qual é muito utilizado em classificação de documentos e modelação de linguagens. É chamado de "*bag*" (que significa saco em português) de palavras, porque qualquer informação sobre a ordem ou estrutura das palavras no documento é descartada. O modelo se preocupa apenas com a ocorrência de palavras conhecida no documento, e não em sua localização. Para perceber como é feita a representação de um documento, pode-se pensar num caso simples, se tem-se um vocabulário de dez palavras e no total dois documentos, a representação de cada documento vai ser um vetor com dez números inteiros, em que cada valor é a ocorrência de cada palavra do vocabulário. Portanto, ao adicionar novos documentos, poderá aparecer novas palavras, o que será necessário aumentar o tamanho dos vetores de cada documento, sendo sequências de zeros. Resultando, assim, em uma matriz esparsa. Outro problema deste tipo de modelo é que não retém qualquer informação gramatical das frases. *Term Frequency - Inverse Document Frequency* (TF-IDF) é um modelo estatístico semelhante ao BoW, mas em vez de calcular a ocorrência das palavras em cada documento, mede a relevância de cada palavra. Para perceber melhor como é calculado esta relevância, é importante em primeiro lugar perceber o que é *Term Frequency* (TF). Observando a equação 2.3, consiste na divisão entre o número de vezes que o termo (t) aparece no documento (d) e o número total de termos que o documento tem (T_d). Desta forma, esta equação mede o quanto o termo é frequente num determinado documento.

$$tf(t,d) = \frac{n(t,d)}{T_d} \tag{2.3}$$

Inverse Document Frequency (IDF) mede o quanto é importante o termo, como se pode ver na equação 2.4, sendo N o número total de documentos e df o número de documentos que contêm t. Portanto, quando o valor de IDF é próximo de zero significa que palavra é pouco relevante, pois o termo aparece na maioria dos documentos.

$$idf(t) = \log \frac{N}{df} \tag{2.4}$$

O valor de TF-IDF resulta da multiplicação de TF com IDF. Ao contrário do modelo BoW que em problemas de casos reais a frequência de ocorrência de certas palavras são valores muito altos, que irá esconder as palavras que são menos frequentes, em TF-IDF isto não acontece.

Distributional Profiles (DP)

DP é um modelo probabilístico que mede a similaridade das palavras através da sua co-ocorrência [36]. Este tipo de modelo, para calcular DP das palavras, utiliza uma janela deslizante que percorre o corpus e caso as palavras se encontrem na mesma janela, é somado à sua contagem das palavras ocorrerem em conjunto, deste modo é construído uma matriz de co-ocorrência. Para medir a força com que as palavras estão associadas, pode ser utilizado probabilidade condicional ou PMI, portanto quanto maior for estes valores, maior será a

tendência de as palavras co-ocorrerem. A partir disto, é possível representar as palavras em vetores de co-ocorrência, ou seja, DP, sendo que se pode utilizar métodos para calcular a distância entre os vetores para medir a sua similaridade, como por exemplo, similaridade do cosseno. Contudo esta abordagem apresenta várias desvantagens, tais como, não é eficaz para palavras que têm vários sentidos, porque irão ocorrer em vários contextos, como por exemplo, a palavra banco pode ser uma instituição financeira ou um objeto para as pessoas se sentarem. Por outro lado, se o vocabulário tem tamanho N, a matriz de co-ocorrência terá tamanho N $\times$ N o que será de grandes dimensões e esparsa.

Word Embeddings Models

Com o desenvolvimento de *deep learning*, tem sido muito utilizado redes neurais para resolver tarefas de PLN, como por exemplo, representação de palavras através das suas características sintática e semântica. Modelos de *word embedding* tem como objetivo quantificar a semântica de cada palavra, não através do significado mesmo da palavra, mas do seu contexto. Esta quantificação é feita por um vetor de valores reais. Cada dimensão do vetor não corresponde a um sentido, mas todas as dimensões em conjunto representam um conceito.

Recentemente, têm sido propostos modelos pré-treinados que conseguem representar uma linguagem universal com processamento de corpus de grandes dimensões, assim não é necessário treinar um novo modelo do zero. O desenvolvimento do poder computacional fez com que aparecesse modelos mais profundos, tornando a fase de treino mais aprefeiçoada. Deste modo, a arquitetura dos modelos passaram de raso para profundo.

Existem dois tipos de modelos de *word embedding*, nomeadamente *non-contextual* e *contextual embeddings* [37].

Começando pelos modelos *non-contextual embeddings*, independentemente do contexto das palavras, o vetor que as representam será sempre o mesmo, ou seja, funcionam como uma tabela de pesquisa. Irá-se analisar alguns dos modelos mais conhecidos, tais como:

- ***Word2Vec:*** modelo de rede neural desenvolvido pela Google [38], que representa cada palavra como um vetor de N dimensões formado a partir das palavras que a rodeia num corpus. Com os vetores aprendidos o modelo consegue explicitamente codificar vários padrões linguísticos, que podem ser demonstrados com translações lineares. Por exemplo, com o seguinte cálculo vec("Madrid") - vec("Espanha") + vec("France") o vetor mais próximo será vec("Paris"), sendo que para uma palavra w, vec(w) corresponde ao seu vetor.

  Word2Vec tem duas arquiteturas de redes opostas, *Continuous Bag-of-Words* (CBOW) e *Skip-gram*. Ambos os modelos têm um hiperparâmetro que consiste no tamanho pré-definido da janela que irá percorrer o corpus para treinar o modelo. CBOW prevê a palavra tendo em conta um conjunto de palavras que representam o seu contexto. Como se pode observar na figura 2.5, CBOW contém três camadas, a *input layer* que diz respeito às palavras do contexto, a *hidden layer* que para cada palavra proveniente da primeira camada atualiza os pesos da matriz, e por fim, *output layer* que corresponde à previsão da palavra alvo. *Skip-gram* é o inverso que o CBOW, ou seja, através de uma

palavra alvo prevê o seu contexto. Ambas arquiteturas têm melhor performance quanto maior a dimensão do corpus, no entanto CBOW é mais rápido na fase de treino e tem melhor performance se houver muitas palavras repetidas no corpus. Caso não se tenha uma grande quantidade de dados de treino e palavras que não sejam muito frequentes, o *skip-gram* revela-se ter melhor performance. Contudo, foram propostos os algoritmos *Negative Sampling* e *Hierarchical Softmax* [39] para melhorar a performance das duas abordagens. *Negative Sampling*, de uma forma breve, durante a fase de treino a rede para cada amostra de dados em vez de atualizar os pesos de todos os neurónios para prever com mais precisão, apenas irá atualizar uma pequena percentagem dos pesos. *Hierarchical Softmax* baseia-se na árvore binária de *Huffman*, que atribui códigos mais curtos às palavras mais frequentes, o que faz com que a fase de treino seja mais rápida. Naili et al. [40] compara CBOW com *skip-gram*, em que os dados de treino são relacionados com a área de informática, o que faz com que se encontre vários termos repetidos. Portanto, a rede neural CBOW apresenta uma melhor eficácia comparativamente ao *skip-gram*, o que se pode concluir que a sua utilização num problema de um contexto específico pode ser útil.

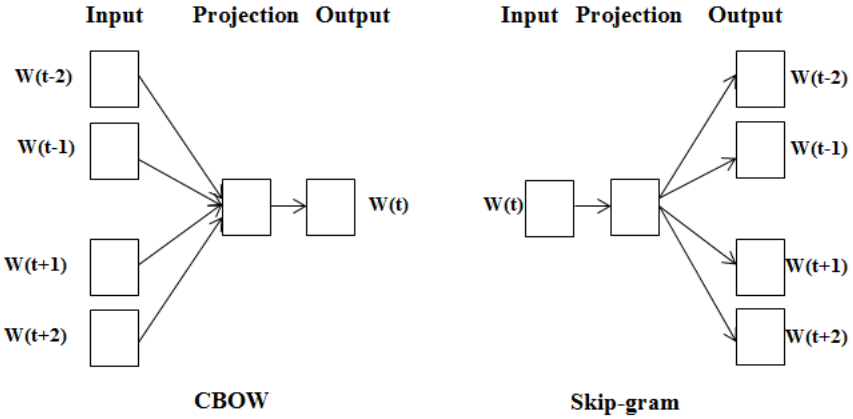


Figura 2.5: Estrutura das redes neurais do modelo *Word2Vec.*[Fonte: [38]]

- ***Global Vectors* (GloVe)**: Pennington et al. [41] propuseram um modelo que tem uma melhor performance que o *Word2Vec*. O *Word2Vec* foca-se essencialmente no conhecimento que se encontra na janela, sendo apenas um contexto local, enquanto que o modelo GloVe baseia-se numa matriz de co-ocorrência global, em que cada elemento Xij da matriz representa a frequência das palavras Wi e Wj que co-ocorreram na mesma janela. No entanto, há estudos que mostram que o modelo *Word2Vec* é melhor, como no caso do trabalho [42], que realizaram várias experiências em que na maioria o *Word2Vec* conseguiu obter melhor performance comparativamente ao GloVe. Portanto, há aqui uma controvérsia de qual o modelo que se comporta melhor, no entanto, quer *Word2Vec* quer GloVe são modelos simples e ambos conseguem aprender a representação semântica das palavras com uma boa eficácia. A principal desvantagem de ambos, é aprender a representar palavras fora do vocabulário, nas quais, podem ser palavras que não

pertencem ao vocabulário atual ou palavras que não apareceram no corpus de treino, o que faz com que o modelo não tenha em conta a parte morfológica das mesmas.

- ***FastText***: Modelo que apareceu mais tarde, desenvolvido pelo Facebook, explicado no estudo [43], com intuito de resolver os problemas dos outros modelos mencionados em cima em relação às palavras fora do vocabulário. *FastText* em vez de alimentar a rede neural com palavras, alimenta com uma parte da palavra, ou seja, parte as palavras em *n-grams* de caracteres, assim consegue criar relações entre as sub-palavras e depois com a soma destes vetores consegue obter a representação semântica da palavra na sua globalidade. Portanto, cada palavra é representada como um *bag* de caracteres de *n-grams*, que são inseridos no modelo *skip-gram* do *Word2Vec*. Os autores afirmam que o seu modelo é rápido, o que lhes permite treinar o modelo com corpus de grandes dimensões e testá-lo com palavras que não aparecem nos dados de treino. Os autores avaliaram o seu modelo em nove línguas diferentes, conseguindo obter bons resultados mesmo com palavras fora do vocabulário na fase de testes. Tendo em conta que, o contexto do nosso problema se trata de um domínio específico, a utilização deste modelo pode não ser muito útil, devido ao facto de que se tivermos um corpus amplo que cobra a maior parte do vocabulário, a probabilidade de se ter uma palavra que não pertença ao vocabulário será muito pequena, o que não se justificaria a sua utilização.

Por outro lado, há palavras que têm mais que um sentido, que dependem do seu contexto para saber o seu significado. Portanto, existem modelos de *contextual embeddings* que recebem sequências de palavras e a partir do contexto da sequência geram um vetor. Por isso, caso a palavra seja polissêmica este tipo de modelos conseguem representar os seus diferentes significados. Iremos agora abordar alguns modelos, tais como:

- ***Embedding from language Models* (ELMo)**: Os modelos de *word embedding* mencionados, como por exemplo, *Word2Vec* e GloVe, são modelos que conseguem capturar informação sintática e semântica das palavras a partir de um texto de grandes dimensões. Contudo, representam a palavra sempre da mesma forma independentemente do contexto, ou seja, para a mesma palavra o vetor resultante será sempre o mesmo. Em 2018, Peters et al. [44] desenvolveram o modelo ELMo que introduz um novo tipo de representação de palavras com uma contextualização profunda, que tem em conta não só a parte sintática e semântica das palavras, mas também como estas variam entre vários tipos contextos linguísticos. Portanto, ELMo para o mesmo token pode obter vetores diferentes caso a palavra se encontre em contextos diferentes. Observando a figura 2.6, pode-se ver que ELMo é constituído por duas camadas *Bidirectional Language Model* (biLM), em que cada uma consiste nas passagens *forward* e *backward*. A entrada da primeira camada biLM são *raw vectors* que foram gerados por uma *Convolutional Neural Network* (CNN) ao nível do caractere a partir de palavras do texto. A representação final é resultante da soma dos pesos do *raw vectors* e dos vetores das palavras intermediárias.

  Para salientar, ELMo tem a vantagem de conseguir representar uma palavra em diferentes vetores consoante o contexto, contudo a sua arquitetura é mais complexa comparada aos outros modelos referidos, o que implica um maior tempo de consumo

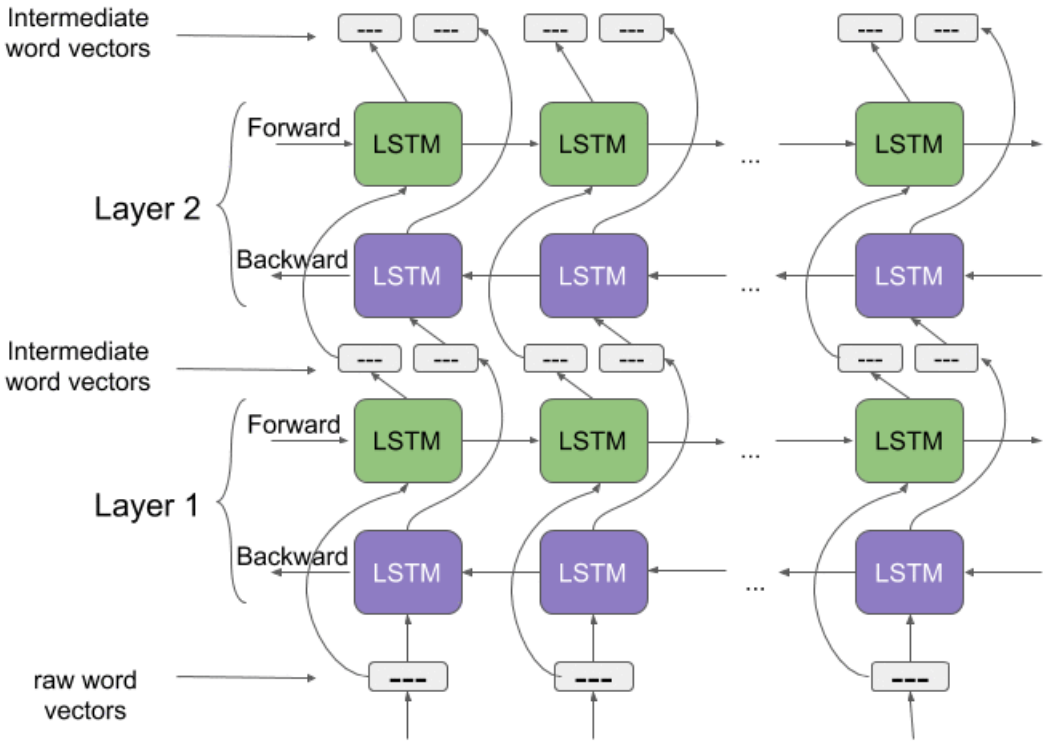


Figura 2.6: Arquitetura do modelo ELMo *Word2Vec.*[Fonte: *Prateek Joshi*]

para gerar os vetores. No entanto, será que compensa esta complexidade adicional, para um tarefa de PLN focada numa área específica onde a variação de contexto é reduzida?

- ***Bidirectional Encoder Representations from Transformers* (BERT)**: No final de 2018, a Google propôs o modelo BERT que conseguiu obter bons resultados em várias tarefas PLN. BERT em vez de utilizar *Recurrent Neural Networks* (RNN) como o ELMo, aplica um modelo de *deep learning*, nomeadamente *Transformer*, que tem a capacidade de percorrer uma entrada de sequência de palavras uma só vez, devido ao facto de não se importar com a ordem com que as palavras se encontram. Como o nome indica, o modelo diz-se que é *bidirectional*, mas na verdade não tem direção, enquanto que as estratégias que utilizam RNN, que é necessário ler as sequências de entrada da esquerda para a direita ou vice-versa. Para treinar o BERT existem duas abordagens diferentes, a primeira chama-se *masked language model* onde 15% dos tokens de forma aleatória serão substituídos pelo token "[MASK]" e o modelo tem que prever os tokens *masked*. A outra abordagem chama-se *next sentence prediction* (NSP), em que o modelo recebe pares de frases e aprende a identificar quando a segunda frase segue-se da primeira. Por fim, o BERT ainda apresenta um rápido *fine-tuning* que é variável dependendo a que tarefa PLN está a ser aplicado. Em 2019, apareceram muitas variantes do modelo BERT que conseguiram obter melhor performance, como por exemplo, RoBERTa [45] onde aplica várias alterações ao modelo BERT, tais como, aumento do tamanho dos *batches* e mais dados na fase de treino, remoção da abordagem NSP, treino de sequências mais longas e alteração dinâmica do padrão de *mask* aplicado aos dados de treino. Estas

alterações fizeram com que o modelo tivesse menos parâmetros e fosse mais eficiente. Para além disso, o ALBERT [46] outro modelo que tentou melhorar o BERT, este que apresenta ter algumas limitações devido ao grande número de parâmetros, o que implica uma degradação do tempo de pré-treino e problemas de memória, por isso, propuseram melhorias, tais como, decompor a matriz de *embedding* do vocabulário em duas matrizes mais pequenas e ter uma *cross-layer parameter sharing* que faz com que se evite que o número de parâmetros não aumente ao longo da profundidade da rede. Estas modificações fazem com que se diminua significativamente o número de parâmetros sem afetar muito a performance do modelo, obtendo assim uma melhor eficácia dos parâmetros. Tendo isto em conta, o ALBERT consegue treinar 1.7 vezes mais rápido que o BERT.

2.4 Trabalhos relacionados

Nas secções anteriores, foram abordados desde de modelos de *word embedding* a redes semânticas e modelos ML para formar relações entre as palavras. Nesta secção, irá-se resumir vários trabalhos da literatura PLN de modo a observar que tipo de combinações podem ser feitas entre *features* e modelos para conseguir obter bons resultados na extração de relações entre palavras e construção de redes semânticas focadas num domínio específico.

Zhang et al. [20] propuseram um modelo para extrair relações de sinónimos que consiste na utilização do *Word2Vec* com uma arquitetura *skip-gram* para treinar um corpus da Wikipédia em inglês. Com isto, aplicaram *Spectral Clustering* e K-means, sendo que obtiveram um F1-score de 0.775 e 0.351 respetivamente. K-means apresenta valores muito inferiores, porque os algoritmos de *clustering* foram aplicados a um *similarity graph* e como tem grande dimensão não conseguiu obter boa performance. Como *spectral clustering* aplica redução de dimensão obteve melhor resultado.

Atzori e Balloccu [14] propuseram um modelo completamente não supervisionado com capacidade de descobrir o hiperónimo dado uma palavra. O modelo consiste em projetar o hipónimo através do modelo *Word2Vec* com uma arquitetura *skip-gram* e encontrar os seus respetivos top-10 vetores vizinhos mais semelhantes utilizando o algoritmo *Nearest Neighbours Search* que é baseado na similaridade do cosseno. Caso haja NP, o modelo trata-os como um vetor resultante da média dos vários vetores de cada token. Com isto, avaliaram o modelo baseando-se na tarefa de descoberta de hiperónimo da competição SemEval 2018, conseguindo ser o modelo não supervisionado com melhor performance. Al-Matham et al. [47] apresentou um modelo semelhante ao anterior, mas para sinónimos em árabe. Realizou uma avaliação de forma a observar a influência do tratamento de NP e *lemmatization*, conseguindo obter resultados melhores, como o caso da medida MAP que alcançou-se uma melhoria de 6.2% e 12.4%, respetivamente. Aplicaram PoS de modo a manter as palavras juntas que tivessem a mesma tag, contudo não surtiu qualquer efeito na performance do modelo.

No entanto, modelos supervisionados apresentam melhores resultados. Shwartz et al. [48] propuseram um modelo com capacidade de detetar entre dois pares se têm uma relação de hiperónimo-hipónimo ou não baseado em LSTMs. Utilizaram redes semânticas, tais como,

WordNet, DBPedia, Wikidata e Yago para extrair várias relações de hiperonímia e técnicas de PLN, como por exemplo, PoS, *lemmatization* e *dependency tree*. Com isto, conseguiram na altura alcançar melhores resultados no estado da arte com um valor de F1-score de 0.901.

Tan et al. [27] é outro modelo que extrai definições do Wikipédia e Stack-Overflow, este é um repositório online onde os programadores procuram respostas aos seus problemas que encontram durante o seu trabalho, o que portanto contém um vocabulário direcionado para *software*. A partir do hiperónimo candidato criam features, tais como, sequências antes e após da palavra baseadas em PoS tags e posição a que se encontram. Com isto, alimentam um GRU *bi-direcional* de modo a aprender padrões, conseguindo alcançar melhores resultados do que métodos existentes em ambos os corpus atingindo um F1-Score 0.913 no corpus da Wikipédia e 0.924 para Stack-Overflow. No entanto, compararam com modelos de linguagem pré-treinados, como o BERT, no qual obtiveram um resultado superior para o corpus Wikipédia com um F1-score de 0.92 e para o corpus Stack-Overflow que apresenta um domínio mais específico, resultado de 0.932.

É importante analisar alguns trabalhos que utilizam as redes CNN em tarefas de PLN, mais especificamente o tipo de arquiteturas que usam. No artigo [49], usam a rede neural profunda para analisar sentimentos e categorização de tópicos, sendo problemas de classificação. As *features* que alimentam a rede são *embeddings* gerados pelo modelo *Word2Vec*, sendo vetores formados a partir da concatenação de palavras de frases compostas. A sua rede tem uma arquitetura simples, na qual é constituída por apenas uma camada convolucional com vários filtros, seguido por uma camada de *max pooling* e, por fim, um classificador *softmax*. Desta forma, usando várias bases de dados utilizadas como referência neste tipo de tarefas, conseguem atingir os resultados de estado de arte. Por outro lado, no trabalho [50] usam uma arquitetura muito mais complexa, com cinco camadas convolucionais contendo diferentes números de filtros, tais como, 1, 32, 64 e 128, e entre cada camada convolucional tem uma camada *max pooling*. Isto para desempenhar a tarefa de semelhança entre frases, que consiste inicialmente em vetorizar as frases através dos modelos BERT e ALBERT. Deste modo, alimentam a rede com bases de dados de referência, conseguindo atingir os resultados de estado de arte, sendo que as representações de ALBERT conseguem obter melhores resultados do que o BERT. Portanto, pode-se realçar que a rede CNN tem a capacidade de obter boas performances em problemas de PLN.

A secção 2.3.3 referiu algumas redes semânticas de domínio específico como a *B-Link* e *TechNet*, mas mais recentemente apareceram novas abordagens. Como no caso da rede que Tóth et al. [51] apresentaram, no qual aplicaram um método para extrair os principais termos utilizados entre desenvolvedores de *software* e criar uma rede semântica baseada em relações de hiperonímia. Utilizaram como base de dados uma coleção de posts do site Stack Overflow. Através dos padrões de Hearst e com a ajuda do *Wordnet* para confrontar alguns resultados, conseguiram construir uma rede representada por um grafo direcionado. Contudo, é uma rede que não consegue cobrir uma grande percentagem de vocabulário, uma vez que ficam dependentes de padrões para formar relações. Chen et al. [52] apresentam outra abordagem para construir a rede semântica de domínio específico de *software*, onde utilizam

FastText devido ao facto de como as palavras extraídas podem conter erros ortográficos, tem em consideração na mesma da informação das sub-palavras e não apenas do contexto. Contudo, em frases longas decompostas em várias sub-palavras, *Fastext* tem dificuldade a encontrar relações entre palavras, por isso combinaram com o modelo *Word2Vec* com uma arquitetura *skip-gram*. Quando dois vetores estão próximos no espaço, utilizam o algoritmo *Damerau-Levenshtein distance* que calcula a *string edit distance*, ou seja, distância entre duas palavras é mínima quando para transformar uma palavra na outra é preciso poucas alterações, isto para determinar se as palavras são morfologicamente semelhantes. Caso contrário, têm um conjunto de regras para verificar se a palavra se trata de uma abreviação. Com isto, conseguiram construir uma rede com 466 228 termos específicos de software em que 442 684 são sinónimos e 18 951 abreviações.

2.5 Sumário

Este capítulo começa por apresentar as várias aplicações de PLN na secção 2.1 e do porquê de ML ter sido adotado em várias tarefas da área.

A secção 2.2 introduz vários passos que são importantes aplicar em tarefas de PLN que trabalham com base de dados textuais. Portanto, é necessário limpar os dados e prepará-los para treinar os modelos ML com o objetivo de que consigam obter um boa performance. Por isso, os primeiros passos são baseados em remover caracteres indesejáveis, realizar uma tokenização para segmentar as frases em tokens, remover palavras insignificantes que pertençam a uma lista de *stop-words* e utilizar *stemming* ou *lemmatization* para reduzir as palavras. Para além disso, o método PoS consegue extrair mais informação das palavras, servindo como *feature* para alimentar modelos de ML ou para encontrar padrões textuais.

Há problemas que nos seus dados existem muitas abreviações, nas quais existem estratégias apresentadas na subsecção 2.2.1 de como se pode detetá-las e substituí-las na sua forma extensa. Classificadores de ML, como por exemplo, SVM, revelem-se ser eficazes para este tipo de problemas, desde que tenham dados suficientes para os treinar. Sendo um passo que pode ser importante no pré-processamento.

Por fim, tem-se o último passo que a subsecção 2.2.2 explica como se pode detetar e tratar de NP, na qual devem ser tratados como um token. O uso da fórmula PMI funciona muito bem para detetar *bigrams*, contudo é limitado para NP constituídos por mais que duas palavras. Muitos trabalhos depois de detetar, concatenam as palavras ou calculam a média dos vetores gerados de cada palavra, como não há uma forma direta para os comparar, a adoção dos dois métodos é uma opção para verificar o desempenho dos modelos de ML.

A secção 2.3 introduz abordagens para criar relações entre as palavras, de como as redes semânticas existentes foram construídas e, por fim, extração de *features*. Portanto, na subsecção 2.3.1 mostra como se pode encontrar relações de hiperonímia e sinonímia através de padrões textuais, porém este tipo de abordagem é ineficiente e está sujeito a ruído textual por natureza. Por outro lado, a subsecção 2.3.2 endereça vários modelos de ML que alcançaram resultados de estado de arte, baseados em aprendizagem supervisionada ou não supervisionada. De seguida, na subsecção 2.3.3 apresenta redes semânticas, como *Wordnet* e *ConceptNet* que cobrem um

vocabulário que não está relacionado com termos da engenharia. Enquanto, as redes *B-Link* e *TechNet* utilizam fontes com termos técnicos e tecnológicos para a construção da rede, o que faz com que cobram um vocabulário de um domínio mais específico. Cada rede tem a sua estrutura e relações diferentes entre as palavras, que podem ser relevantes para a expansão de vocabulário. Para finalizar, tem-se a subsecção 2.3.4 que endereça vários modelos para extrair *features* muito utilizados para alimentar modelos de ML. Na subsecção 2.3.4, apresenta o modelo BoW que representa cada documentos em vetores do tamanho do vocabulário, sendo cada valor o número de ocorrências da palavra no documento. Enquanto que TF-IDF, mede a relevância das palavras. Na subsecção 2.3.4 fala sobre DP que é um modelo probabilístico, no qual mede a similaridade das palavras através da sua co-ocorrência. Por fim, os modelos de *word embedding* descritos na subsecção 3.4.5 que são muito utilizados para resolver tarefas de PLN, nos quais são divididos em duas categorias. *Non-contextual embeddings* que são modelos simples e conseguem representar as palavras de forma eficiente, contudo representam cada palavra sempre com o mesmo vetor, mesmo que se mude o contexto. Enquanto que, os *contextual embeddings* apresentam modelos que resolvem este tipo de problema, embora sejam mais complexos e custosos computacionalmente a treinar.

Por fim, a secção 2.4 descreve vários trabalhos relacionados com o contexto desta dissertação, para observar que tipo de combinações podem ser feitas entre *features* e modelos para conseguir relacionar as palavras de uma forma eficaz.

CAPÍTULO 3

Solução Proposta

A ideia principal da dissertação é desenvolver um modelo que consiga encontrar relações de semelhança entre termos de redes e de *software* de modo a que possa ser útil para mapear diagnósticos de alarmes para um modelo único. Para a resolução do problema, tem-se como base uma quantidade considerável de *logs* reais de eventos de alarme de dispositivos de redes, o que numa fase inicial será necessário utilizar várias técnicas de paralelização e de pré-processamento de forma a limpar o ruído dos dados textuais e prepará-los para serem modelados.

Na literatura apresentada, existem vários modelos para extrair *features*, como por exemplo, os modelos pré-treinados de *word embedding* que conseguem representar uma linguagem pelas características semânticas e sintáticas das palavras. Portanto, o objetivo é utilizar estes modelos para projetar cada palavra do vocabulário no espaço, ou seja, as palavras serão convertidas em vetores e através da distância do cosseno é possível medir a similaridade. Este cálculo quanto mais próximos dois vetores estiverem no espaço, maior será o seu valor de similaridade.

No entanto, a nossa solução consiste em adicionar uma camada com modelos de ML de modo a ajustar esta medida. Os modelos devem aprender uma nova função de similaridade através de um conjunto de pares de vetores, com intuito de preverem as similaridades com mais rigor entre termos pertencentes a um vocabulário específico de redes e *software*. Para que isto seja possível, tem-se uma segunda base de dados construída manualmente por um *captcha* de palavras, que consiste em pares de palavras encontradas por especialistas da área que são sinónimas ou não. A ideia é que cada modelo de *word embedding* transforme os pares de palavras em pares de vetores para alimentar os modelos de ML. Isto permite encontrar relações entre as palavras do vocabulário com mais precisão.

3.1 Modelos de *word embedding*

Há uma diversidade de modelos de *word embedding* que se pode utilizar. Um muito popular é o BERT, um dos modelos que na literatura alcançou resultados de estado de arte em

tarefas de PLN, contudo contém milhões de parâmetros o que implica ter muitos recursos de *hardware* para conseguir treiná-lo. Para além disso, tem a capacidade de representar palavras polissémicas, sendo que para a mesma palavra pode devolver vetores diferentes consoante em que contexto se encontra, o que para o nosso problema, como é relacionada com uma área específica, este tipo de situação não é relevante. Por isso, recorreu-se a modelos independentes do contexto como o *Word2Vec* e *FastText*, que são modelos mais leves e menos complexos comparativamente ao BERT e como o corpus de treino tem uma tamanho considerável, se torna mais fácil em questões de recursos e de tempo para conseguir obter modelos capazes de extrair *features* das palavras.

3.2 Modelos de ML

No estado de arte na secção 2.3.2, são mencionados vários tipos de redes neurais que são muito utilizadas em problemas de PLN e que através das representações vetoriais das palavras conseguem obter bons resultados relativamente à extração de relações. Selecionou-se três redes, tais como, a CNN, LSTM e rede neural simples. A CNN é uma rede neural profunda que consegue lidar muito bem com matrizes e como usa filtros que percorrem os dados matriciais, pode ser eficaz na aprendizagem. A LSTM também é uma rede profunda e complexa que, como tem a capacidade de aprender dependências de longo prazo, pode encontrar padrões importantes que podem influenciar o valor de similaridade entre os vetores, sendo relevante para o nosso problema. Será utilizado a rede neural simples para efeitos de comparação com as outras redes de modo a verificar como a complexidade interfere nos resultados.

3.3 Arquitetura

Para se perceber melhor a solução na sua globalidade, criou-se uma imagem com a arquitetura geral do sistema. Como se pode ver na figura 3.1, a partir da base de dados, onde contém os diagnósticos dos alarmes, é feito inicialmente um pré-processamento de forma paralela, ou seja, cada *worker* recebe uma parte dos dados e executa a *pipeline* para limpar e processar os seus dados, que depois guarda-os em disco em um ficheiro. A partir destes ficheiros, são treinados os modelos *Word2Vec* e *FastText*, nos quais têm como função de representar cada palavra em vetores de N dimensões. Por fim, a partir dos vetores e da base de dados criada manualmente, alimenta-se os modelos de ML de forma a que sejam capazes de prever se duas palavras são semelhantes ou não.

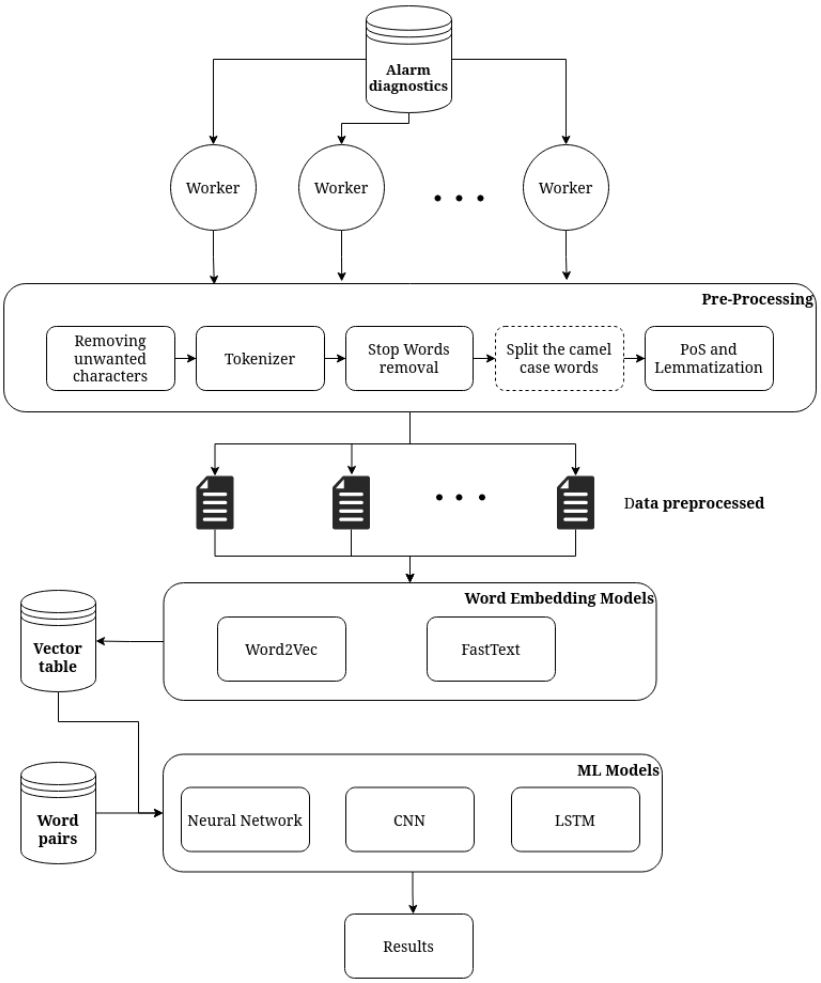


Figura 3.1: Arquitetura geral da solução proposta.

3.4 Implementação

Há muitos algoritmos implementados disponíveis que permitem com que o desenvolvimento do sistema seja mais fácil e rápido. Essencialmente na área de ML que os modelos são complexos e requer alguns recursos computacionais, o que precisam de estar otimizados. Há bibliotecas que contém os modelos implementados, o que faz com que se poupe muito tempo na fase de desenvolvimento. Este capítulo irá abordar as várias ferramentas que foram utilizadas, como se implementou a solução proposta e problemas que se enfrentou durante o desenvolvimento.

3.4.1 Ferramentas

No desenvolvimento dos módulos foi utilizado a linguagem de programação *Python* [53] pelo facto de oferecer muitas ferramentas que são úteis para o processamento de dados, uso de modelos de *word embedding*, implementação de modelos de ML e apresentação de resultados. Portanto, nesta secção irá-se descrever as principais bibliotecas utilizadas, tais como:

- **Numpy** [54] fornece estrutura de matrizes e vetores de N dimensões que apresenta várias funções matemáticas que torna a sua utilização rápida e versátil. A sua sintaxe

é de alto nível o que torna intuitivo durante a escrita de código. Esta biblioteca é importante para o processamento de dados.

- **Re (RegEx)** é uma biblioteca em Python que fornece operações para encontrar expressões regulares. As operações permite-nos procurar padrões no texto a partir da definição de uma sequência de caracteres. Este módulo facilita em técnicas de processamento de dados, como por exemplo, na limpeza de caracteres indesejáveis.
- **Natural Language Toolkit (NLTK)** [55] é um kit de ferramentas prático, simples e bom para fins educacionais. Fornece vários métodos, tais como, tokenização, PoS para atribuir tag às palavras, *stemming* e *lemmatization* para reduzir as palavras. Para além disso, ainda permite transferir listas de *stop words*. NLTK permite facilmente implementar uma *pipeline* típica de pré-processamento de uma tarefa PLN.
- **Gensim** [1] disponibiliza modelos de *word embedding* já implementados, como por exemplo, *Word2Vec*. Estes modelos estão otimizados para que consigam treinar de forma rápida e independente da memória, ou seja, está pronto para treinar corpus de grandes dimensões sem precisar de carregá-lo todo em memória RAM. Sendo, que consegue processar um grande volume de dados através de métodos preparados para *data streaming*.
- **Scikit-learn** [56] é uma biblioteca em Python que fornece ferramentas simples e eficientes para ML e modelagem estatística, como por exemplo, classificação, regressão, redução de dimensão e entre outros.
- **TensorFlow** [57] é uma plataforma completa que facilita a criação e a implantação de modelos de ML. Através da sua API *Keras* de alto nível, facilmente se cria os modelos. Esta biblioteca é importante para o desenvolvimento dos modelos ML para definir as relações entre as palavras.
- **MatPlotLib** [58] é uma biblioteca abrangente para criar visualizações estáticas, animadas e interativas em Python. Disponibiliza métodos que facilmente se constrói gráficos, sendo útil para a apresentação de resultados.

3.4.2 Base de dados

Relativamente aos dados, foi fornecido pela empresa Nokia uma base de dados privada em inglês de eventos de alarmes de dispositivos de rede reais de diferentes operadoras, que é importante para a construção do vocabulário e para treinar os modelos de *word embedding*. A sua estrutura consiste em vários ficheiros, onde cada um corresponde a um conjunto de diagnósticos de alarmes associados a uma rede, que aconteceram num determinado dia do ano. Cada diagnóstico é formado a partir de diferentes campos, que nos dá informação relativa ao alarme. No topo de cada diagnóstico surge de imediato a identificação do dispositivo, o código do alarme e as horas em que sucedeu. De seguida, encontra-se campos que nos fornece informação sobre o estado, tipo e causa do alarme. Para além disso, há um campo com o nome *Additional Text* que descreve o problema em texto mais decorrido, o que para a vetorização das palavras é importante, visto que permite saber o contexto em que as palavras estão inseridas. Embora haja diagnósticos que possam ter campos diferentes consoante a que

[1] https://radimrehurek.com/gensim/index.html

rede pertença, a informação mais importante encontra-se em todos nos mesmos campos, ou seja, é possível encontrar um padrão para extrair a informação necessária. A base de dados tem 69 Gib de tamanho, o que implica ter em conta técnicas de multiprocessamento para acelerar os processos de pré-processamento.

Foi necessário criar uma base de dados para treinar e avaliar a performance dos modelos de ML que desempenham a função de associar relação de semelhança entre duas palavras, ou seja, uma palavra pode ser substituída pela outra, sem mudar o sentido da frase. Para isso foi necessário criar uma página web que segue a ideia de *captcha* de palavras, ou seja, o site consiste em um quiz em que o utilizador tem que selecionar a expressão, que pode ser uma palavra ou mais, semelhante à expressão que aparece na pergunta. Sendo, que há uma opção para caso de nenhuma das expressões serem semelhantes com a expressão alvo. As perguntas do quiz eram construídas a partir de grupos de palavras semelhantes calculados a partir dos modelos de *word embedding* (mais detalhado na secção 3.4.5), onde escolhia-se de forma aleatória um das três estratégias para selecionar as expressões. Uma das estratégias consistia em escolher uma palavra aleatória do vocabulário como expressão alvo e as palavras que se encontram no mesmo grupo que a mesma, apareciam na parte das respostas. Outra estratégia era selecionar as expressões de forma aleatória independentemente do grupo a que pertencesse, pelo facto de puder aparecer pares semelhantes que estivessem em grupos distintos. O último critério baseava-se em repetir perguntas que foram menos respondidas para aumentar o número de votos nos respetivos pares de modo a aumentar a confiança da amostra.

A partir dessas respostas é calculado a similaridade entre as duas expressões através da fórmula 3.1, em que no denominador encontra-se o número de vezes que as palavras A e B ocorreram juntas na mesma pergunta, ou seja, contabiliza-se quando uma expressão aparece na pergunta e a outra na parte das respostas independentemente da resposta do utilizador. Na parte do numerador, tem-se a variável P que é um contador da força das respostas, o que portanto sempre que o utilizador selecionar que a expressão A é semelhante à expressão B é incrementado 1 à variável P, caso o utilizador selecionasse a opção *None*, ou seja, a expressão alvo que aparece na pergunta não é semelhante a nenhuma das expressões que aparecem nas respostas, desincrementava-se 1 à variável. Deste modo, o valor da similaridade varia entre -1 e 1 e quanto mais perto o valor de similaridade for de um, mais similares serão as duas palavras.

$$sim(expressionA, expressionB) = \frac{counter(P)}{total(expressionA, expressionB)} \tag{3.1}$$

Com isto, cada instância da base de dados consiste em pares de vetores, resultantes da projeção feita pelos modelos de *word embedding* dos pares de palavras, e a sua respetiva similaridade calculada a partir das respostas. Com base nestes dados, são posteriormente utilizados para treinar e testar os modelos de ML. Conseguiu-se obter no total de 954 respostas de vários especialistas e de alunos universitários da área de redes. Como se pode ver na figura 3.2, tem-se 421 respostas *None* e 533 que não, ou seja, os utilizadores escolheram uma expressão que aparecia na parte das respostas.

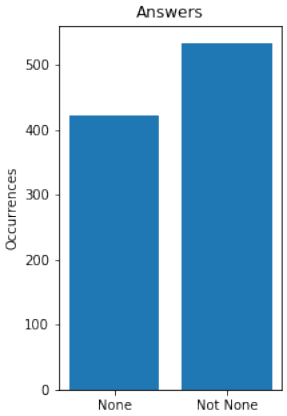


Figura 3.2: Gráfico relativo às respostas obtidas pelo *captcha*.

Tendo em conta que o vocabulário tem mais de duas mil palavras, a amostra obtida pelas respostas não é suficientemente grande para cobrir todo o vocabulário. Por outro lado, os modelos de ML requerem uma grande quantidade de dados para se obter bons resultados, o que portanto pode-se dizer que esta base de dados é limitada.

3.4.3 Pré-processamento

Primeiramente realizou-se uma seleção de dados apresentados nos diagnósticos, isto porque existia campos que continham informação sobre localizações, datas, identificações de dispositivos, endereços Internet Protocol (IP), identificadores que são irrelevantes para este trabalho e são confidenciais. Após esta seleção, implementou-se a *pipeline* típica de pré-processamento muito utilizada em problemas de PLN, que foi apresentada na secção 2.2. Esta *pipeline* foi implementada com ajuda da biblioteca *NLTK* [2], na qual contém várias funções de processamento da linguagem natural em *Python*. Com isto, removeu-se todo o tipo de caracteres que não são letras para depois efetuar a tokenização de espaço em branco, ou seja, dividiu-se as frases pelos espaços, tabs e caracteres especiais, isto é caracteres que representam quebra de linha ou pertencem a uma lista específica de tokens que devem ser removidos. A biblioteca tem outros tipos de métodos de tokenização que encontram expressões regulares específicos no texto para fazer a divisão, mas para este trabalho não é relevante. Outro passo importante, é remover todo o tipo de tokens que pertença à lista de *stop words* de palavras inglesas de modo a diminuir o ruído dos dados. Para além disso, como nos diagnósticos de alarmes existem muitos termos escritos em *camel case*, como por exemplo, *CommunicationsAlarm*, utilizou-se a biblioteca de *Python*, nomeadamente *re*[3], que consiste em métodos para operações em expressões regulares encontradas, para detetar este tipo de situação e dividir em duas palavras ou mais, no exemplo referido ficaria *Communications Alarm*.

Por fim, usou-se o método *lemmatization* para reduzir as palavras na sua forma raiz. Optou-se por este método em vez de *stemming*, por ter demonstrado ter uma melhor performance e,

[2]https://www.nltk.org/
[3]https://docs.python.org/3/library/re.html

por isso, que vários trabalhos apresentados no estado de arte escolheram esse tipo de método. Como é mais rigoroso a cortar as palavras, ou seja, tem em conta a morfologia das palavras, garante-nos que não forme várias raízes para a mesma palavra ou pelo menos o erro seja mínimo. Sendo assim, a biblioteca *NLTK* fornece-nos este módulo implementado, no entanto, para que o *lemmatization* faça a redução, é necessário que se aplique PoS de modo a criar as tags para cada palavra, pois isto ajuda o método a perceber a parte morfológica das palavras.

Como o tamanho do ficheiro de dados de diagnósticos é significativo, o módulo de pré-processamento foi implementado de forma a ser multiprocessado e não carregar todos os dados em memória. Portanto, dependendo do número de *cores* que é indicado ao módulo, é criado o mesmo número de *workers*, ou seja, um *worker* é uma função que irá aplicar a *pipeline* à quantidade de dados que recebe. O resultado de cada *worker* é guardado em disco, quando todos os *workers* finalizarem o seu trabalho, obtém-se, assim, um conjunto de ficheiros que contêm os dados pré-processados.

3.4.4 Vocabulário

Após os dados terem sido pré-processados, o próximo passo consiste em definir o vocabulário que irá conter as palavras para criar relações entre si. Embora se tenha selecionado campos dos diagnósticos, onde a maioria da informação é relativa a dispositivos de redes, e eliminado palavras que pertencem à lista de *stop words*, não é o suficiente para obter um vocabulário específico de redes e *software*. Há muito ruído que é preciso ser removido.

Então, decidiu-se calcular os valores de TF-IDF que representam a relevância de cada palavra, ou seja, caso uma palavra ocorra muita vez, é menos relevante, o que é muito provável ser ruído e deve ser excluída. Desta forma, ao remover este tipo de termos, terá-se um vocabulário mais limpo e com um domínio específico consistente.

No desenvolvimento deste módulo utilizou-se a biblioteca *Scikit-Learn*, que contém a classe *TFIDFVectorizer* [4]. Esta permite-nos extrair *n-grams*, o que para este trabalho é importante para detetar *bigrams*, ou seja, duas palavras que ocorrem no texto muita vez juntas, na quais podem ser consideradas como um NP, o que deve ser tratado como um token. Com isto tem-se o valor TF-IDF de cada palavra, onde se obteve um vocabulário de 229718 palavras, que se ordenarmos por ordem decrescente, pode-se imaginar graficamente uma curva descendente. A ideia é usar métodos que sejam capazes de encontrar pontos ideais na curva, nomeadamente joelhos, onde se possa fazer um corte para separar as palavras que apresentam valores TF-IDF baixos, das palavras com valores altos. Assim, ao remover todas as palavras com valores baixos, fica-se com as palavras mais relevantes que serão o vocabulário relacionado com redes e *software*. Para este processo, utilizou-se algoritmos para detetar joelhos, tais como, *Kneedle*, o princípio de *Pareto* e *L-Method*.

Kneedle

O algoritmo *Kneedle* [59] usa o conceito de curvatura como uma medida matemática de forma a medir o quanto uma função difere de uma linha reta. A partir disto, o algoritmo

[4] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

tenta encontrar o ponto ou joelho da função, ou seja, a parte da função onde a curva se dobra visivelmente, na qual é uma zona que a função passa de alta inclinação para baixa inclinação (plana ou planalto). *Python* tem uma biblioteca, nomeadamente *kneed*[5], que torna possível calcular o *knee point* da função com os valores da frequência de TF-IDF através da classe *KneeLocator*. Esta classe tem o parâmetro S que permite ajustar a sensibilidade ou agressividade do algoritmo para encontrar os pontos de joelho. Portanto, quanto menor for o valor de S, mais rápido se deteta o joelho.

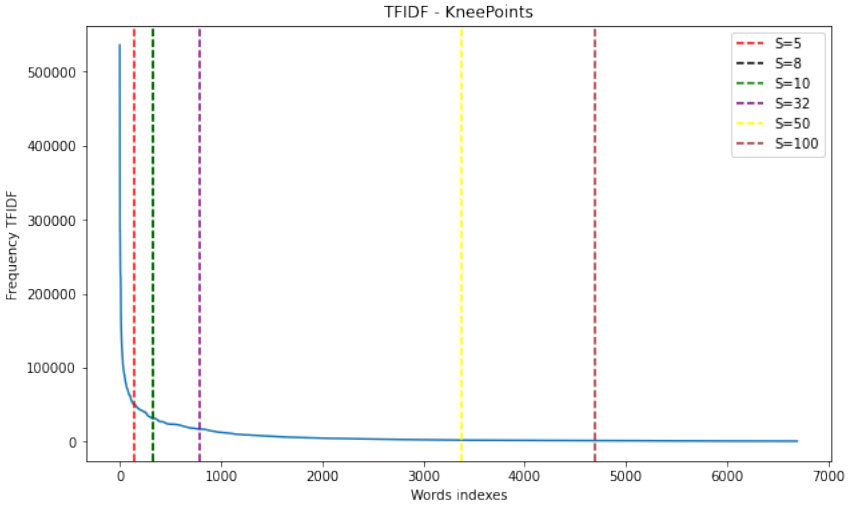


Figura 3.3: Gráfico que apresenta a frequência de TF-IDF das palavras e os pontos joelhos de possíveis cortes que se pode realizar na função calculados a partir do algoritmo *Kneedle*.

Na figura 3.3, é visível a curva da frequência de TF-IDF e através dos tracejados verticais os vários cortes que o algoritmo calculou consoante o valor do parâmetro de sensibilidade.

Príncipio de Pareto

O princípio de Pareto ou regra de 80/20 [60] afirma que 80% das consequências são produzidas através de 20% das causas. Na prática, pode ser visto como um gráfico com duas funções, uma que representa as frequências relativas das palavras ordenadas por ordem decrescente da esquerda para a direita e a outra função representa a percentagem cumulativa. Na figura 3.4, pode-se observar que é feito um corte através da reta vertical laranja a tracejado, que representa o instante em que a função cumulativa de percentagem alcança os 80%. Portanto, com o método do princípio de Pareto permite-nos obter um vocabulário com 2569 palavras.

[5]https://kneed.readthedocs.io/en/stable/

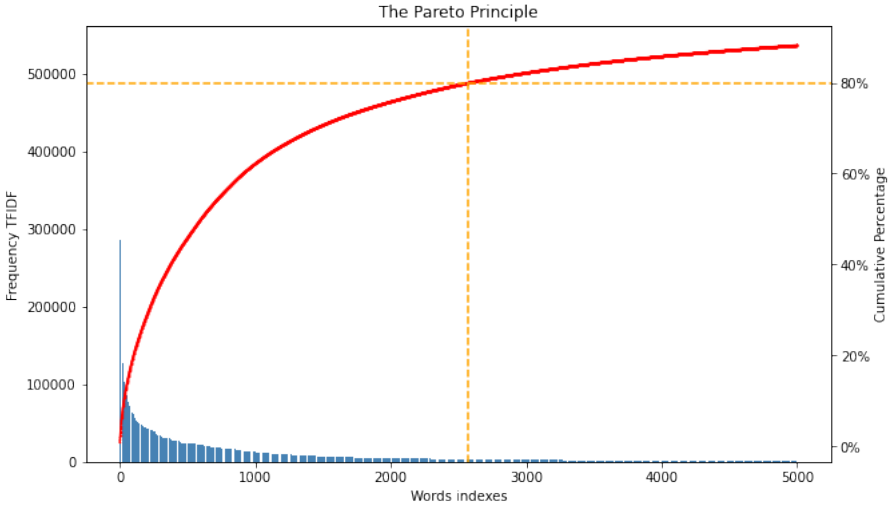


Figura 3.4: Gráfico que apresenta a frequência de TF-IDF das palavras e corte da função calculado a partir do princípio de Pareto.

L-Method

O algoritmo L-Method [61] tem o mesmo objetivo que o algoritmo *Kneedle*, isto porque procura também encontrar o ponto de joelho na função. A diferença é que neste método o ponto é calculado de forma diferente. Em L-Method traça-se duas retas, em que uma tenta cobrir o máximo de pontos do lado esquerdo do gráfico, onde se encontra uma grande inclinação e a segunda reta tenta cobrir os pontos do lado direito da função, que correspondem à zona de planalto. A intersecção entre as duas retas é o ponto de joelho. Utilizou-se o algoritmo já implementado [6] e o ponto de joelho calculado encontra-se no ponto 1978.

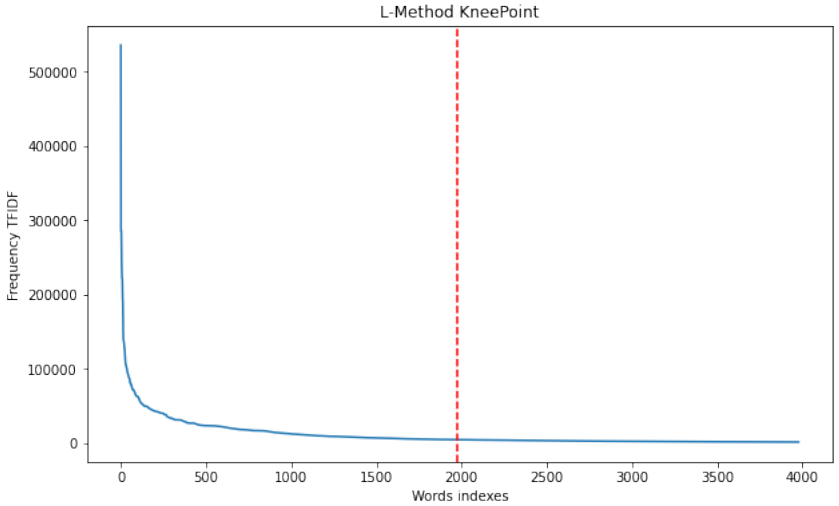


Figura 3.5: Gráfico que apresenta a frequência de TF-IDF das palavras e corte da função calculado a partir do L-Method.

[6] https://github.com/mariolpantunes/knee/blob/main/knee/lmethod.py

Discussão

Como se pode ver, cada método obteve o seu ponto de corte, no qual são todos distintos. Os resultados do algoritmo *Kneedle* pode-se verificar que para valores pequenos do parâmetro de sensibilidade, o corte é muito agressivo, como por exemplo, na figura 3.3 pode-se observar que para S igual a cinco, oito e dez são cortes que atingem a função entre a parte de grande inclinação e o planalto. Enquanto, os outros valores de S atingem vários pontos diferentes do planalto. Portanto, é verificável a implicação que a variável sensibilidade tem na deteção dos pontos de joelho.

Na figura 3.4, pode-se ver que com o princípio de Pareto que o corte obtido, atinge o início do planalto da função, isto porque como as primeiras frequências são valores muito altos, comparativamente às frequências das outras palavras, então atinge os 80% da frequência rapidamente tendo em conta a frequência total. Portanto, a soma das frequências das 2569 palavras são 80% da frequência total de todas as palavras. O que demonstra que existe muitas palavras que não são relevantes, nas quais são ruído que devem ser excluídas do vocabulário.

Na figura 3.5, observa-se o corte em 1978 calculado pelo L-Method, que comparativamente ao princípio de Pareto o corte é mais perto da curva que se encontra entre a parte de grande inclinação da função e o planalto. Isto porque, como este algoritmo desenha uma reta que tenta ficar sobreposta ao maior número, a reta na parte do planalto deve seguir a mesma inclinação, enquanto que na parte de grande inclinação como é só algumas palavras que tem uma frequência alta, a reta deve seguir a inclinação da curva para cobrir o máximo de pontos. Caso a reta tivesse a inclinação do declínio inicial da função, o ponto de joelho atingiria a curva, sendo o corte muito agressivo.

O objetivo de utilizar esses métodos é reduzir o máximo possível de ruído, de modo a reter as palavras relacionadas com redes e *software*. Um corte ideal na função, tendo em conta o objetivo dos algoritmos, é encontrar o ponto na curva que está entre o planalto e a zona de alta inclinação, mas seria um corte muito agressivo. Portanto, é preferível fazer um corte na zona do planalto da função, que embora possa-se obter algum ruído, cobre o máximo de palavras da área específica, do que fazer um corte agressivo e cortar palavras que pudessem ser importantes. Por isso, há quatro cortes calculados que atingiam o planalto, dois deles foram calculados pelo princípio de Pareto e L-Method, que atingem a função nos pontos 2569 e 1978, respetivamente. Os outros dois foram calculados pelo algoritmo *Kneedle*, um no ponto 3804 para S igual a cinquenta e o outro no ponto 4695 para S igual a cem. Decidiu-se utilizar o algoritmo de joelho com um S igual a cinquenta, por comparativamente aos outros pontos, não é tão agressivo o que garante que cobre todas as palavras relevantes e não é tão conservador, o que não adiciona tanto ruído como o ponto calculado com S igual a cem. Com isto, obteve-se um vocabulário com 3379 palavras, que agora é necessário projetá-las no espaço para que seja possível criar relações entre as mesmas.

3.4.5 Extração de *features*

A extração de *features* é uma fase do trabalho onde acontece a transformação das palavras pré-processadas em vetores. O objetivo desta fase é que cada palavra do vocabulário seja

representada por um vetor para que depois se possa criar pares de palavras, que serão pares de vetores que servirão para alimentar os modelos de ML.

A linguagem *Python* disponibiliza a biblioteca *Gensim*, onde contém o *Word2Vec* e *FastText* implementados. A biblioteca permite-nos guardar os vetores gerados pelos modelos de *word embedding* para que depois se possa transformar os pares de palavras encontrados através do *captcha* em pares de vetores. Para além disso, antes de cada treino é necessário que cada modelo saiba quais as palavras que devem ser tratadas como NP, o que é possível detetar através da classe *Phrases*[7] que depois serão tratados como um *token* pelos modelos. Na tabela 3.1, observa-se alguns dos parâmetros que são apresentados pela classe *Phrases*.

Parâmetros	
min_count	1
threshold	5
scoring	default

Tabela 3.1: Parâmetros da classe *Phrases*.

Este algoritmo utiliza uma função de pontuação dos *bigrams* baseado no artigo [62] que utiliza a fórmula 3.2, na qual baseia-se nas contagens de *unigram* e *bigrams*, na variável *min_count* que corresponde ao mínimo de vezes que as duas palavras devem ocorrer e, por fim, a variável *len_vocab*, ou seja, o tamanho do vocabulário. Efetivamente, este cálculo é mais simples que o de PMI, embora a ideia seja a mesma, quanto mais as palavras ocorrerem em conjunto, maior será a pontuação. Caso a pontuação ultrapasse o valor de *threshold*, o *bigram* passa a ser considerado como um NP.

$$\frac{(bigram\_count - min\_count) * len\_vocab}{(wordA\_count * wordB\_count)} \tag{3.2}$$

É importante salientar que *Gensim* tem uma classe com o nome *PathLineSentences* que permite treinar os modelos a partir de dados guardados em disco. Como a base de dados tem um tamanho considerável, é sempre uma preocupação de como se irá carregar os dados, pois é impossível carregar todos os dados em memória RAM, a não ser que se tivesse muitos recursos de *hardware*. Então, tem-se uma diretoria onde contém os ficheiros com todos os dados pré-processados e, portanto, o caminho relativo para o programa chegar a esta diretoria será passado como parâmetro na classe *PathLineSentences* e esta responsabiliza-se por ir carregando os dados à medida que são necessários, sem que tenhamos que preocupar com a memória RAM. Desta forma, é possível treinar os modelos de uma única vez.

Word2Vec

O *Word2Vec*, como foi referido anteriormente, através do contexto das palavras que ocorrem no corpus de treino, representa-as em vetores de N dimensões, em que o seu principal objetivo é agrupar vetores de palavras semelhantes num espaço vetorial.

[7] https://radimrehurek.com/gensim/models/phrases.html

Após a deteção de *bigrams*, os dados de treino são introduzidos na classe *Word2Vec*[8] do *Gensim* para treinar com uma janela de tamanho cinco que irá percorrer o corpus, sendo a partir deste contexto que irá gerar um vetor para cada palavra. Neste projeto, é relevante analisar alguns hiperparâmetros do modelo, se conseguem melhorar ou não a performance dos modelos de ML. Primeiramente, o *Word2Vec* tem duas arquiteturas opostas, nomeadamente CBOW e *Skip-gram*, que é importante perceber com a quantidade de dados de treino e frequência das palavras, qual o tipo de arquitetura irá obter melhores resultados. E, ainda, foram propostos os algoritmos Negative Sampling (NS) e Hierarchical Softmax (HS) para melhorar a performance do modelo, que através da biblioteca *Gensim* é possível ativar estes métodos durante a fase de treino. Como referido no estado de arte, o NS durante a fase de treino atualiza apenas uma parte dos neurónios da rede neural e HS atribui códigos mais pequenos às palavras mais frequentes. Deste modo, espera-se que estes métodos acelerem o tempo de treino. Há outros hiperparâmetros que podem melhorar a performance, tais como, as épocas que são o número de iterações que o corpus será submetido e o tamanho dos vetores. A variação de parâmetros encontra-se na tabela 3.2, que será feita para cada modelo, ou seja, *Word2Vec* com a arquitetura *Skip-gram* e CBOW, e por fim, o modelo *FastText*. O parâmetro *vector_size* é referente ao tamanho dos vetores que os modelos irão gerar, o parâmetro *hs* é referente ao algoritmo HS e o parâmetro *negative* tem haver com o algoritmo NS. É importante realçar que caso o HS e NS sejam zero, significa que não será aplicado nenhum destes algoritmos durante a fase de treino dos modelos. Quando o HS é igual a um, o algoritmo é ativado. Em relação ao NS, segundo a documentação quando o valor é maior que zero, o algoritmo é ativado, mas normalmente utiliza-se valores entre cinco a vinte, sendo que este valores representam a quantidade de palavras que a rede irá atualizar os pesos. A influência deste tipo de valor no modelo não será um objetivo desta dissertação, apenas será aplicado com o valor igual a dez.

Parâmetros	
vector_size	128, 256, 384, 512
hs	0, 1
negative	0, 10

Tabela 3.2: Parâmetros utilizados para os modelos *Word2Vec* e *FastText*.

Quando as representações das palavras estiverem calculadas, cada palavra do vocabulário está armazenada em estrutura de uma tabela de consulta ou dicionário, o que facilmente consegue-se obter os vetores.

FastText

O modelo *FastText* é um modelo mais recente do que o *Word2Vec*, que tem a capacidade de vetorizar palavras que estão fora do vocabulário. Isto porque, parte as palavras em *n-grams* para alimentar a rede neural. Deste modo, as relações são encontradas entre conjunto de

[8]https://radimrehurek.com/gensim/models/word2vec.html

caracteres e não entre palavras completas. Isto pode ser importante, porque na criação do vocabulário tem-se em conta NP que contém no máximo duas palavras, o que durante a fase de testes pode aparecer *bigrams* que podem não ter sido detetados pelo módulo *Phrases* do *Gensim*, que devem ser vetorizados. Para além disso, o *FastText* é um modelo rápido, o que consegue treinar um corpus de grandes dimensões em pouco tempo.

A biblioteca *Gensim* tem o modelo *FastText*[9] implementado e na sua documentação pode-se ver que tem vários *hiperâmetros*. O estudo dos mesmos será semelhante ao modelo *Word2Vec*, para depois se fazer uma comparação entre os dois tipos de modelos. O *FastText* e o *Word2Vec* do *Gensim*, na sua documentação, apresentam um método chamado "most_similar", no qual retorna as dez palavras mais próximas no espaço de uma certa palavra e as suas respetivas similaridades. O valor de similaridade é medido a partir da distância de cosseno entre dois vetores, o que varia entre -1 e 1. Com isto, criou-se grupos de palavras semelhantes para ajudar a gerar o *captcha* de palavras como foi referido na secção 3.4.2. Portanto, percorreu-se todas as palavras do vocabulário e para cada uma utilizava-se o método "most_similar", sendo que das palavras retornadas eram inseridas no mesmo grupo de sinónimos, se tivessem mais que 0.6 de similaridade. No entanto, havia casos em que uma palavra pertencia a mais que um grupo, nesta situação a palavra era inserida no grupo, onde tivesse maior valor de similaridade. Desta forma, era mais provável que as palavras que aparecessem nas respostas do quiz do *captcha* tivessem relacionadas com as palavras alvo, assim era uma forma mais eficiente de encontrar pares de palavras do que gerar as questões apenas escolhendo palavras do vocabulário de forma aleatória. Para além disso, como o *Gensim* permite guardar os modelos de *word embedding*, será útil para calcular similaridades entre pares de palavras e comparar com as similaridades calculadas a partir das respostas do *captcha*.

3.4.6 Modelos de ML

Esta secção irá-se descrever a última fase deste projeto, na qual o seu principal objetivo é medir a similaridade entre as palavras através das suas representações vetoriais. Começou-se por implementar uma rede neural simples com apenas três camadas e, posteriormente, uma CNN e LSTM que são redes mais profundas. Os modelos foram construídos com a ajuda da interface *Keras*, onde contém as camadas necessárias para a implementação dos mesmos. Dependendo do modelo, têm hiperparâmetros diferentes, que podem influenciar a sua performance.

Para treinar estes modelos, usou-se a base de dados construída manualmente que está descrita na secção 3.4.2. Com as 954 respostas recolhidas, conseguiu-se criar 2076 exemplos que serão os dados de entrada dos modelos de ML. Cada exemplo é constituído por dois vetores que são um par de palavras em que foi calculado a sua similaridade apresentado como terceiro parâmetro, que deverá ser o valor expectado pelos modelos. Portanto, os modelos serão treinados para regressão e com uma aprendizagem supervisionada em que todos os modelos na última camada é utilizada uma função de ativação *Sigmoid* que faz com que o o

[9] https://radimrehurek.com/gensim/models/fasttext.html

valor de saída é entre zero e um. Dentro deste intervalo, o valor quanto mais perto de um, mais similares serão os pares de palavras e mais perto de zero, menos similares são.

Na figura 3.6, pode-se observar a arquitetura da rede neural mais simples que irá servir de comparação para com as redes mais profundas. A primeira camada e a segunda utilizam função de ativação ReLU, sendo a função muito utilizada neste tipo de camadas por melhorar a performance do modelo [63]. Por fim, tem-se a terceira camada com apenas um nó, que corresponde a um valor entre zero e um. Para este modelo, irá-se analisar como as épocas poderá alterar o seu desempenho.

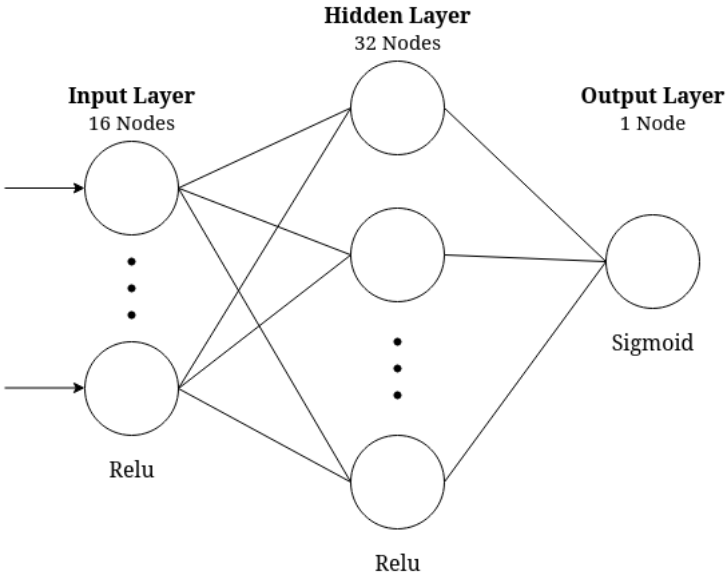


Figura 3.6: Arquitetura da Rede Neural.

Na figura 3.7, como se pode ver é uma rede mais complexa que a anterior, onde se tem duas camadas convolucionais, que têm 32 e 64 filtros, respetivamente, e ambas com a função ReLU de ativação. Entre as duas camadas encontra-se uma camada de *Max-Pooling* com uma janela de tamanho 2×2. A seguir da segunda camada convolucional, tem-se uma camada de *Flatten* que tem como objetivo em transformar uma matriz bidimensional em um vetor apenas, que os seus valores servirão de entrada para a camada com 128 nós. Por fim, na última camada tem-se apenas um nó que irá conter o valor final. No trabalho [49] para classificação de frases, conseguiram obter bons resultados com uma CNN com apenas uma camada convolucional, porém é difícil basear-se em arquiteturas de outros trabalhos, porque o problema e os dados são diferentes. Portanto, na criação da arquitetura optou-se por não se ter muitas camadas, para não ser uma rede muito complexa, pelo facto que o modelo pode basear-se muito a sua aprendizagem nos dados de treino, que posteriormente na fase de testes poderá não conseguir obter boas previsões, sendo um caso de *overfitting*. Nas redes convolucionais é possível definir o tamanho dos filtros que irão percorrer a matriz de dados, sendo um hiperparâmetro que poderá ser importante para o seu desempenho. Como a entrada da rede será uma matriz de $2 \times N$, sendo duas linhas por ser a representação de duas palavras e N o tamanho do vetor de cada palavra, inicialmente definiu-se os filtros com um tamanho de 1×3.

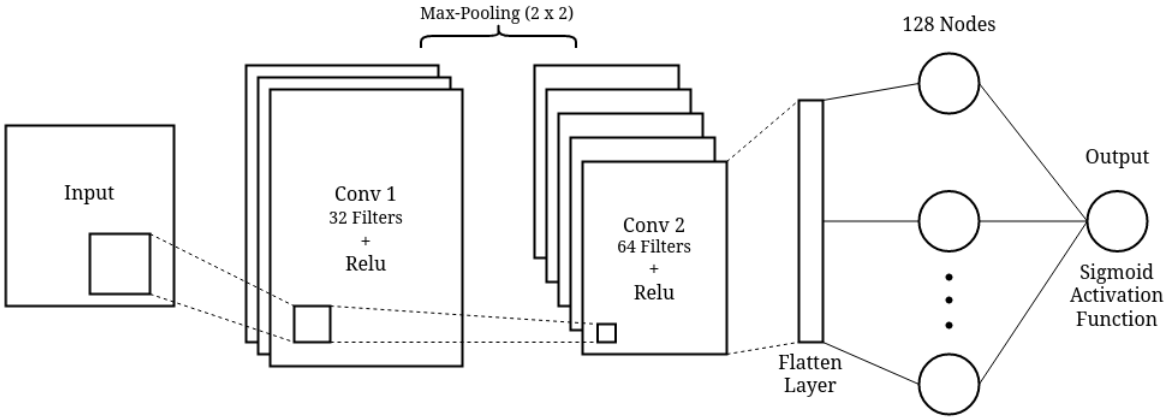


Figura 3.7: Arquitetura da CNN.

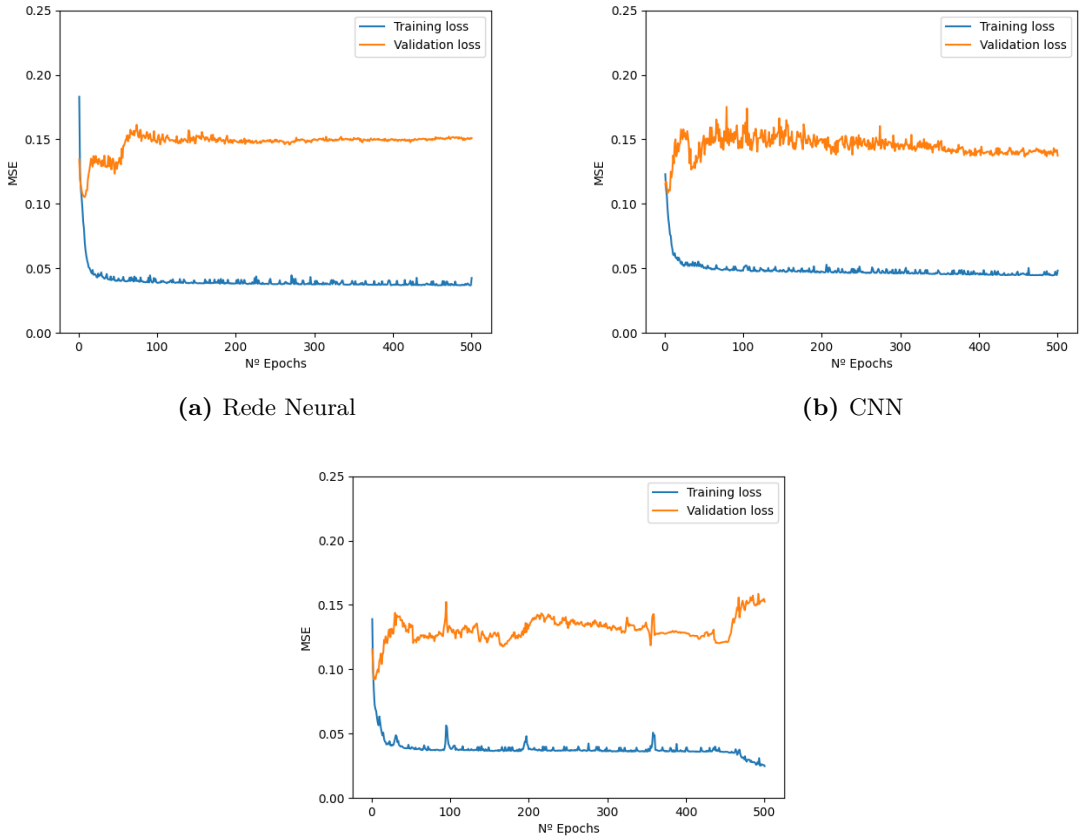
Para o desenvolvimento de uma rede baseada em LSTM, definiu-se uma arquitetura em que a primeira camada é uma camada LSTM com 300 unidades de memória e retorna a sequência, ou seja, a próxima camada irá receber a sequência e não apenas valores. A segunda camada é outra camada LSTM com o mesmo número de unidades de memória. A seguir de cada camada LSTM, aplica-se uma camada de *dropout*, no qual é uma camada que de forma aleatória ignora uma percentagem de certos nós da rede durante a fase de treino, de modo a prevenir o problema de *overfit*. Neste caso usou-se este tipo de camada com uma percentagem de 20%, ou seja, irá ignorar 20% dos nós. Por fim tem-se uma camada toda conectada com um função *Sigmoid* de ativação com apenas um nó, que terá o valor final.

Os modelos de ML não se pode simplesmente treiná-los com os dados de treino e esperar que consiga ter uma boa exatidão com dados que não apareceram na fase de treino. Por isso, para avaliar a estabilidade dos modelos realizou-se *cross-validation*, ou seja, a partir dos dados de treino e de teste, em cada época, isto é, cada passagem que o modelo faz completa no conjunto de dados de treino, validou-se o modelo no estado em que se encontra. Este processo serve para garantir que o modelo não tenha aprendido muito os detalhes e ruído dos dados de treino e consiga fazer boas previsões com novos dados.

Sendo assim, dividiu-se a base de dados gerada pelas respostas obtidas pelo *captcha* em dois conjuntos, um tem 90% dos dados para treinar os modelos e o outro 10% dos dados para testar e validar. Esta proporção foi escolhida pelo facto de que a base de dados não é muito extensa e se a percentagem dos dados para treino fosse menor, seria poucos dados para os modelos conseguirem aprender. Com os dados de teste, irá-se medir a performance do modelo através do cálculo do erro a cada estado dos modelos baseado na diferença entre o valor real e o valor que foi previsto, sendo necessário definir uma função de custo. Esta função estima o erro do modelo a cada época e o objetivo é que durante a fase de treino, os pesos possam ser atualizados de forma a reduzir o erro na próxima avaliação. Como o nosso problema se trata de uma regressão, utilizou-se a função MSE, no qual é uma função muito usada neste tipo de problemas.

Então, a partir do modelo *Word2Vec* com uma arquitetura de *skip-gram* e vetores de 256 dimensões, aplicou-se o *cross-validation* para verificar o comportamento dos modelos de ML.

Na figura 3.8, pode-se observar que de forma geral os modelos na fase de treino obtiveram de erro por volta de 0.05, enquanto que na fase de validação, o erro é muito maior, sendo uma diferença de 0.10. Este tipo de situação acontece quando estamos perante um caso de *overfitting*.



(a) Rede Neural

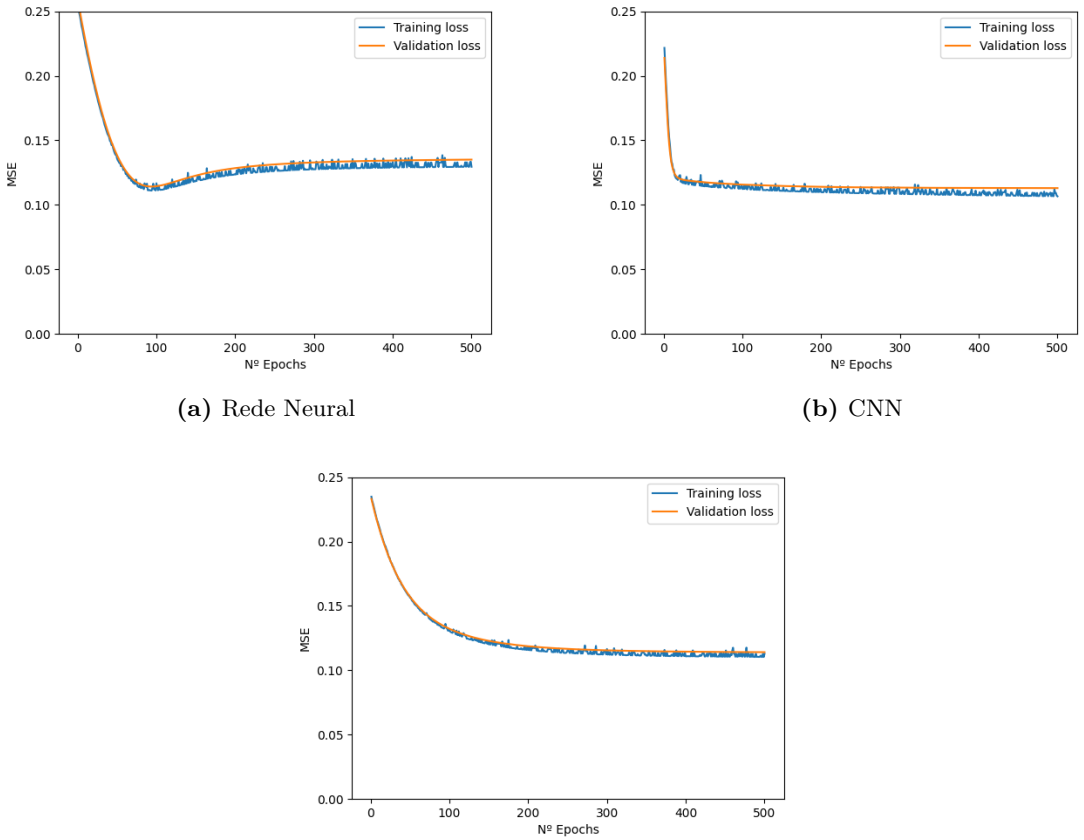
(b) CNN

(c) LSTM

Figura 3.8: Primeira tentativa de *cross-validation* dos modelos de ML com dados gerados pelo modelo *Word2Vec* com uma arquitetura *skip-gram* e vetores de 256 dimensões.

Para resolver este problema, há várias formas, tais como, reduzir a complexidade da arquitetura dos modelos, aumentar número de dados de treino, remover algum tipo de dados que possa estar a criar ruído e entre outros. A primeira modificação feita foi em alterar o otimizador de Adam para SGD, apesar de que o Adam seja um algoritmo que convirja mais rápido, o SGD consegue generalizar melhor o erro, o que pode influenciar o resultado final da performance do modelo.

Por isso, voltou-se a realizar o *cross-validation* dos modelos com o otimizador SGD, sendo que ainda diminuiu-se o seu *learning rate* de 0.01 para 0.001, isto é quanto maior o seu valor, mais rápido o algoritmo tenta chegar ao mínimo da função de custo. Contudo, pode não encontrar por ir demasiado rápido, por isso diminuiu-se o seu valor. Deste modo, irá demorar mais tempo a encontrar o mínimo, mas é muito provável que o modelo consiga convergir. Com isto, na figura 3.9 pode-se observar os resultados do *cross-validation* dos modelos de



(a) Rede Neural

(b) CNN

(c) LSTM

Figura 3.9: *Cross-validation* dos modelos de ML com otimizador SGD e dados gerados pelo modelo *Word2Vec* com uma arquitetura *skip-gram* e vetores de 256 dimensões.

ML, de forma geral verifica-se que em todos os modelos as retas relativas ao erro de treino e de validação estão praticamente sobrepostas, em vista disso pode-se afirmar que o caso de *overfitting* foi resolvido. Por outro lado, o erro de treino aumentou e o de validação diminui, sendo que a CNN e LSTM ao longo das épocas o valor de erro tende para 0.10, sendo visível uma melhor estabilidade nos modelos. A rede neural o erro tende para 0.14, embora tenha melhorado consideravelmente a sua performance, ainda poderá ser possível melhorar mais ainda, pelo facto que, na figura 3.9a, é visível na época 90 que o modelo encontrou-se num estado onde obteve melhores resultados, com valores de erros próximos de 0.10, mas que de seguida aumentou lentamente para 0.14. Em vista disso, tentou-se diminuir o erro de validação através do aumento de complexidade do modelo, ou seja, adicionar mais camadas à rede neural de modo a melhorar a sua aprendizagem. Sabendo que isto faz com que proporcione o *overfit*, acrescentou-se camadas de *dropout* por prevenção. Através da arquitetura com uma camada inicial de 128 nós que irá receber os dados de treino, quatro camadas intermediárias cada uma com 128 nós e uma camada de *dropout* com uma percentagem de 20% e, por fim, a última camada com um nó que corresponde ao valor final, conseguiu-se obter uma rede melhor que a anterior, com um valor de erro a tender para os 0.10 como a CNN e LSTM, no qual se pode verificar na figura 3.10.

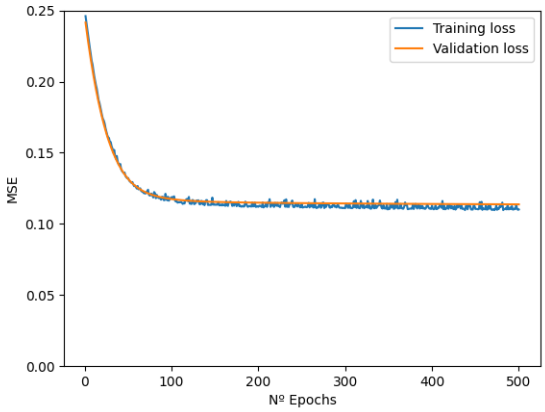


Figura 3.10: *Cross-validation* da rede neural com uma arquitetura mais complexa.

Por fim, foi necessário fazer algumas alterações nas métricas utilizadas durante o treino dos modelos de ML. Estes como o último nó utiliza uma função de sigmoid que retorna valores entre 0 e 1, tem uma escala diferente que os modelos de *word embedding*, ou seja, se fosse aplicado as métricas MAE e MSE não se poderia comparar entre a solução proposta com os modelos nativos de *word embedding*, porque se a escala for menor, o valor de erro seria naturalmente menor também. Portanto, para ignorar as escalas e para efeitos de comparação, utilizou-se funções de correlação, tais como, coeficiente de correlação de Pearson e correlação de classificação de Spearman. Este tipo de funções medem o quanto estão dois conjuntos de dados relacionados, ambas retornam valores entre -1 e 1, nos quais o valor 1 representa uma relação positiva perfeita e o valor igual a -1 representa uma relação negativa perfeita e, por fim, se for 0 significa que não têm qualquer tipo de relação. Na equação 3.3, mostra como o coeficiente de Pearson é calculado, sendo X_i o valor do conjunto de dados X, Y_i o valor do conjunto de dados Y e n o número de observações. O coeficiente de correlação de Pearson é muito usada para relações lineares entre duas variáveis distribuídas normais.

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2(y_i - \overline{y})^2}} \tag{3.3}$$

Na equação 3.4, mostra como é calculado a correlação de classificação de Spearman, onde n é o número de amostras e d_i representa a distância aos pares dos valores X_i e Y_i. Este coeficiente ao contrário com o de Pearson, não interessa a distribuição das variáveis.

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2 - 1)} \tag{3.4}$$

Portanto, a ideia é os modelos preverem as similaridades entre os pares de palavras que se encontram nos dados de teste e conjuntamente com as similaridades "reais", calcular os coeficientes de correlação. Desta forma, se estes dois conjuntos de dados estiverem fortemente relacionados, significa que os modelos conseguiram prever valores maiores de similaridade quando dois termos eram suposto ser semelhantes e valores baixos quanto os termos não eram semelhantes.

Para treinar as redes neurais utilizou-se como métrica o coeficiente de correlação de Pearson, que era calculado a partir de uma função [10] implementada em Keras. Para além disso, como as redes durante a fase de treino ajustam os pesos dos seus nós de forma a conseguirem diminuir o erro, é necessário alterar a função de custo para uma mais ajustada de modo a aumentar os valores de correlação e a melhorar a aprendizagem dos modelos. A função implementada em Keras consiste em calcular o erro em função do coeficiente de correlação de Pearson, ou seja, o erro é igual a $1 - r^2$, sendo r o coeficiente. Deste modo, pode-se afirmar que quanto maior for o coeficiente, menor é o erro. Assim, os modelos a cada iteração conseguem ter uma aprendizagem mais ajustada ao que se é avaliado.

Com estas alterações, pretende-se que os modelos de ML a partir de pares de palavras representadas em vetores de N dimensões geradas pelos modelos de *word embedding* sejam capazes de medir as similaridades entre os termos do vocabulário específico o mais próximo possível das similaridades esperadas.

3.5 Sumário

No início deste capítulo é apresentado a solução proposta para medir a similaridade entre palavras de vocabulário relacionado com redes e *software*. A solução consiste inicialmente em processar uma base de dados composta por diagnósticos de alarmes de forma a reduzir o máximo de ruído, para que seja possível treinar modelos de *word embedding*. Estes modelos têm a capacidade de converter as palavras em vetores numéricos. Portanto, através de uma segunda base de dados composta por pares de palavras gerados a partir de um *captcha* de palavras, pretende-se utilizar os modelos de *word embedding* para transformar estes pares de termos em pares de vetores para alimentar os modelos de ML, com intuito de ajustar as similaridades.

De seguida, é descrito as ferramentas utilizadas para o desenvolvimento da solução proposta. Para além disso, na secção 3.4.2 descreve-se as duas bases de dados que foram utilizadas, uma é composta por diagnósticos de alarmes de dispositivos reais de redes e a outra foi gerada a partir de um *captcha* de palavras com intuito de encontrar termos semelhantes dentro do vocabulário, sendo que cada instância contém os vetores dos pares de palavras e a sua respetiva similaridade calculada a partir das respostas. No entanto, esta base de dados é limitada por não se ter obtido respostas suficientes para se ter uma amostra que garanta confiança.

Posteriormente, na secção 3.4.3 demonstra-se todos os passos realizados de pré-processamento aos diagnósticos de alarmes de modo a remover o máximo ruído possível. Deste modo, tinha-se um vocabulário com 29718 palavras. Por isso, foi necessário calcular o TF-IDF que mede a relevância das palavras e com o uso de algoritmos de joelho, apresentados na secção 3.4.4, foi possível encontrar um ponto de corte no vocabulário que reduziu-o para 3379 termos.

A partir deste vocabulário, na secção 3.4.5, descreve-se como se treinou os diferentes modelos de *word embedding*, nomeadamente o *Word2Vec* e *FastText*, que têm como função

[10] https://github.com/WenYanger/Keras_Metrics/blob/master/PearsonCorr.py

representar os termos do vocabulário em vetores numéricos. Portanto, os pares de palavras encontrados nas respostas obtidas pelo *captcha* converteu-se cada palavra do vocabulário em vetores. Desta forma, na secção 3.4.6, mostra como a as redes neurais, a CNN e LSTM irão receber estes dados nas suas redes e conseguirem treinar de maneira a prever as similaridades o mais próximo possível das similaridades medidas a partir do *captcha*. As arquiteturas iniciais dos modelos sofreram alterações de modo a tornar as redes mais estáveis e a combater com o problema de *overfitting*. Por fim, teve-se que ajustar as funções de custo das redes, devido às métricas a que são avaliadas durante o treino, tais como, coeficientes de correlação de Pearson e Spearman. Estas métricas foram usadas para que seja possível depois fazer uma comparação dos modelos de ML com os modelos de *word embedding*.

CAPÍTULO 4

Resultados

A solução proposta utiliza diferentes modelos de *word embedding* e de redes neurais. Portanto, neste capítulo será descrito as experiências que foram aplicadas a estes modelos de forma a que se consiga perceber os resultados em relação às suas respetivas performances. Assim, permite-nos fazer uma comparação entre eles e perceber quais os que têm maior potencial na resolução do problema desta dissertação. Foram treinados múltiplos modelos de *word embedding* para gerar vetores de diferentes tamanhos que servem para alimentar os modelos ML. Com a avaliação dos mesmos, consegue-se perceber qual a melhor combinação para calcular similaridade entre pares de palavras relacionados com vocabulário de redes e *software*.

4.1 Dados

Na secção 3.4.2 descreveu-se as duas bases de dados utilizadas nesta dissertação, a primeira tem um tamanho significativo que foi utilizada para treinar os modelos de *word embedding* após ser feito o pré-processamento. E, ainda, construiu-se o vocabulário, no qual contém 3379 palavras, depois de ter sido feito o corte com o algoritmo de joelho com o parâmetro de sensibilidade igual a cinquenta. A segunda base de dados descrita foi construída manualmente a partir das respostas obtidas pelo *captcha* para avaliar os modelos de ML, o que será feito uma análise mais a fundo dos pares de palavras gerados, pois como a quantidade de respostas não é considerável, poderá influenciar nos resultados finais. Portanto, tinha-se dito anteriormente que das 954 respostas, 533 foram pares de palavras e não a opção *None*. Agora focando-se apenas nas 533 respostas, tendo em conta que pode haver respostas iguais, averiguou-se que apenas 393 pares de palavras foram encontrados. Este conjunto de pares de palavras corresponde a 639 palavras diferentes, o que em 3379 palavras no total pertencentes ao vocabulário, com um cálculo simples equivale a 18.9% de cobertura do vocabulário. Isto, pode ser porque o vocabulário é esparso ou o número de respostas não foi o suficiente. Outro ponto importante é o número de vezes com que os pares de palavras foram votados. Como se pode

ver na figura 4.1, dos 393 pares de palavras encontrados, 299 foram respondidos apenas uma vez, sendo muito raro haver pares que foram votados mais que três vezes.

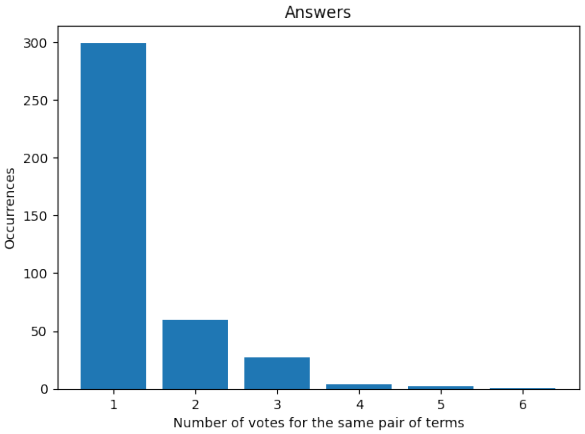


Figura 4.1: Gráfico sobre a quantidade de vezes que os pares de palavras encontrados foram votados pelos especialistas da área.

Em relação aos pares criados pode-se afirmar não se tem amostras de respostas suficientes para que se possa garantir com certeza que um determinado par de termos são realmente semelhantes. Por isso, estes pares de palavras foram revistos manualmente por técnicos da área de forma a aumentar a confiabilidade dos dados, onde houve vários ajustes no valor das similaridades entre termos.

Com base nas respostas foi gerado uma base de dados com 2096 instâncias, que será utilizada para treinar e avaliar os modelos de ML. Cada exemplo trata-se de três parâmetros, os dois primeiros são os vetores que representam o par de palavras e o último o valor da similaridade. O número de exemplos consegue-se obter mais do que o número de respostas, porque, tendo em conta a fórmula 3.1, quando a resposta é *None* são criadas relações entre a palavra alvo e as palavras que se encontram nas opções da resposta, contudo contribui negativamente para o cálculo da similaridade. Portanto, a base de dados é desequilibrada pelo facto de que a maioria dos exemplos estão associados a similaridades negativas. Para se compreender o tipo de palavras que existem no vocabulário e tipo de relações que foram criados a partir do *captcha*, criou-se um grafo, como se pode ver na figura 4.2, com alguns exemplos de pares de palavras. Esta pode-se observar que cada nó corresponde a um termo e a ligação entre os nós, caso seja verde escuro representa uma forte ligação, se for um verde mais claro, demonstra uma ligação menos forte e, por fim, se a ligação tiver uma cor vermelha, é porque têm uma similaridade negativa. No conjunto de termos apresentado pode-se encontrar várias abreviações que não foram transformados na sua forma extensa, pelo facto que uma abreviação pode ser vista como um NP, então seria representada como um token pelos modelos de word *embedding*. Pois, na fase de treino, os modelos iriam ter em conta o mesmo contexto quer a abreviação estivesse na sua forma extensa ou não, obtendo assim a mesma representação.

Por outro lado, é importante realçar que as similaridades calculadas a partir das respostas do *captcha* e ajustadas por especialistas, serão vistas como os valores corretos, nos quais os modelos de *word embedding* e de ML devem conseguir expectar. Na fase de testes, a

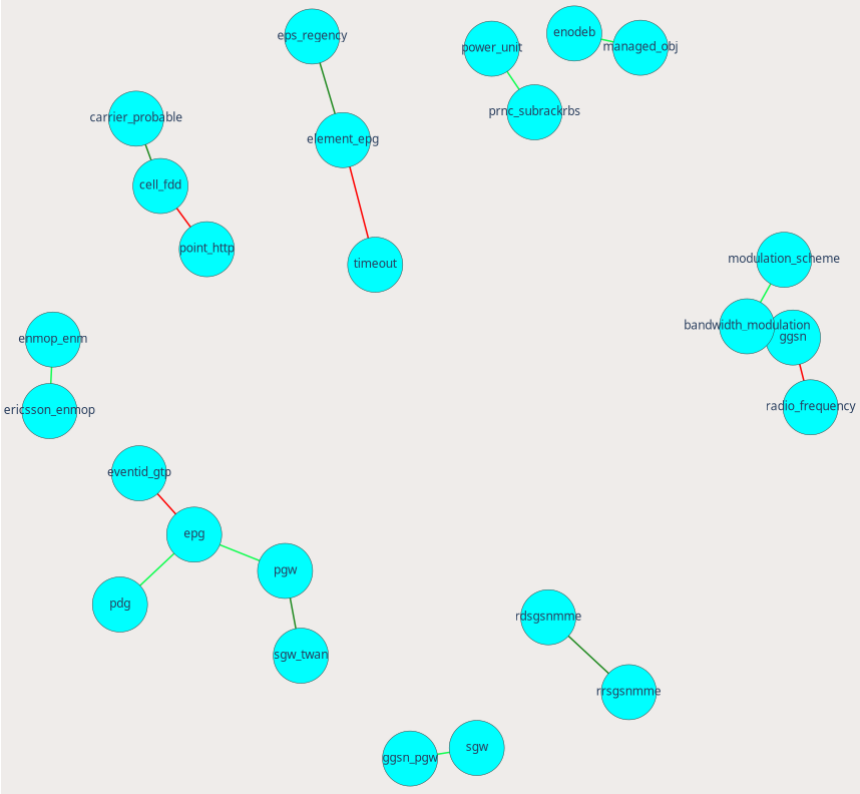


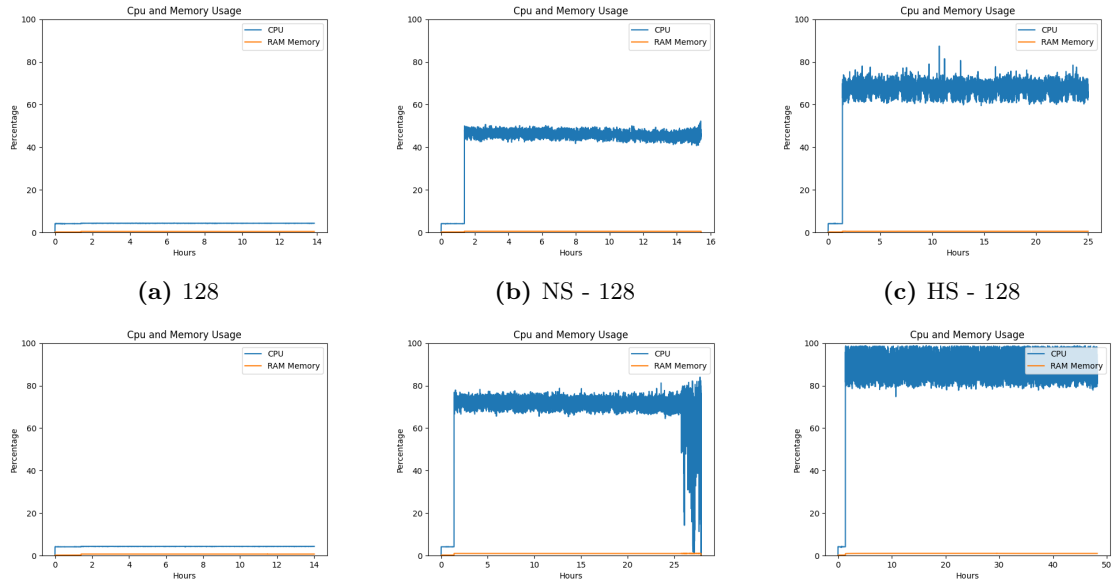
Figura 4.2: Exemplos de pares de palavras encontradas por especialistas da área de redes e *software*.

base de dados será dividida em duas partes, uma parte será para treinar os modelos de ML e a outra para testar não apenas os modelos de ML, como também os modelos de *word embedding*. Como se tem poucos dados, escolheu-se uma percentagem de 90% dos dados para a fase de treino e 10% para testes, igual como se fez durante o *cross-validation*. Embora, haja métodos implementados que fazem esta divisão automaticamente a partir de diferentes critérios, preferiu-se implementar um método personalizado com finalidade de escolher os pares mais votados para teste, ou seja, os 10% de dados de testes serão pares de palavras que ocorreram mais vezes nas respostas. Desta forma, a avaliação dos modelos será mais precisa.

4.2 Modelos de Word Embedding

O *Word2Vec* e *FastText* são modelos leves que conseguem ser treinados num computador com poucos recursos, enquanto que o BERT já não é possível, porque necessita de muitos recursos computacionais. No entanto, utilizou-se uma máquina virtual com 32 GBs de memória RAM e 24 vCPUs para treinar os modelos o mais rápido possível. Deste modo, treinou-se vários tipos de modelos com diferentes parâmetros que se pode observar na tabela 3.2. Para se perceber melhor a leveza dos modelos, com a ajuda da biblioteca psutil[1] implementou-se um processo que foi executado de forma paralela à fase de treinos dos modelos, que tem como função de guardar cinco em cinco segundos a percentagem de utilização de CPU e de memória

[1] https://psutil.readthedocs.io/en/latest/



(a) 128 **(b)** NS - 128 **(c)** HS - 128

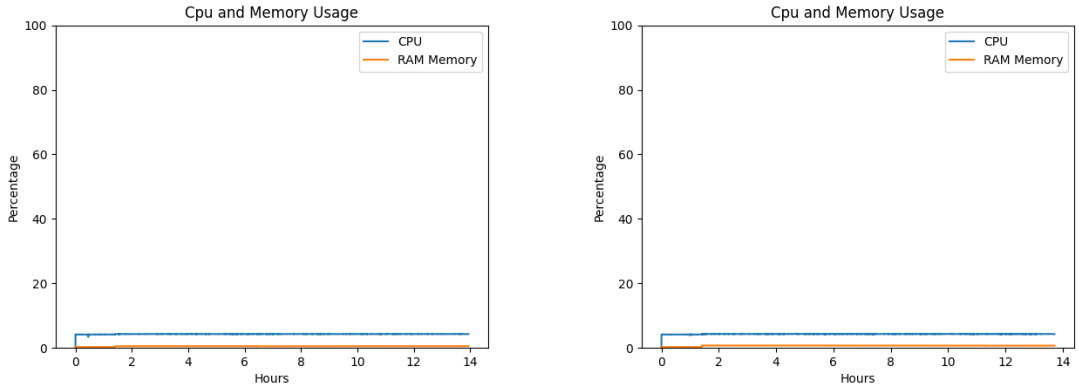
(d) 512 **(e)** NS - 512 **(f)** HS - 512

Figura 4.3: Uso de CPU e memória RAM durante o treino de *Word2Vec* com arquitetura *skip-gram* com vetores de tamanho igual a 128 ou 512.

RAM. Na apêndice A.1 encontra-se a média destas métricas medidas durante a fase de treino dos modelos de *word embedding* e o tempo total que demoraram a ser treinados.

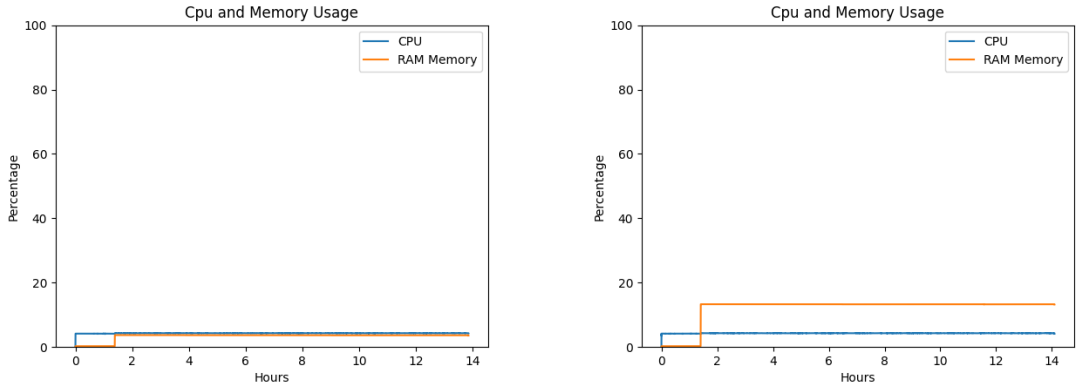
Inicialmente, treinou-se o modelo *Word2Vec* com a arquitetura de *Skip-gram*. Em todas as figuras apresentadas em 4.3, no início de cada gráfico, os modelos utilizam poucos recursos, porque definem o vocabulário antes de começarem a treinar, o que computacionalmente é uma operação leve. Em relação à memória RAM utilizada, quer seja utilizado NS ou HS ou nenhum dos dois, não existem diferenças, como referido na secção 3.4.5, o gerenciamento da memória é feito através do módulo *PathLineSentence* da biblioteca Gensim que carrega os dados quando é necessário, como se pode ver, as percentagens de utilização de memória RAM são muito pequenas. No entanto, em relação à utilização do CPU, se observarmos as linhas azuis, na figura 4.3a e 4.3d, observa-se valores muito baixos, enquanto que os modelos com NS e HS obtiveram valores muito superiores, sendo que NS utiliza menos CPU que HS. Por exemplo, na figura 4.3b usa entre 40 a 50% de CPU enquanto que na figura 4.3c usa entre 60 a 80%. O que faz sentido, na medida que NS não atualiza todos os pesos das camadas da rede, o que tem menos custo computacional. Entre dimensões de vetores, se compararmos os gráficos 4.3b com 4.3e e 4.3c com 4.3f, verifica-se que quanto maior for o vetor, maior é a utilização de CPU. Contudo, entre as figuras 4.3a e 4.3d, não importa o tamanho do vetor, que a utilização de recursos computacionais é igual. Por fim, o tempo de treino é maior para vetores com maiores dimensões e com a utilização dos algoritmos NS e HS, o que por questões de tempo, treinou-se os próximos modelos sem estes algoritmos.

Portanto, de seguida treinou-se modelos *Word2Vec* com arquitetura CBOW sem os algoritmos NS e HS, gerando vetores de diferentes dimensões. Na figura 4.4, observa-se que são gráficos semelhantes aos gráficos 4.3a e 4.3d, ou seja, há um uso reduzido de memória RAM



(a) 128 **(b)** 512

Figura 4.4: Uso de CPU e memória RAM durante o treino de *Word2Vec* com arquitetura CBOW com vetores de tamanho igual a 128 ou 512.



(a) 128 **(b)** 512

Figura 4.5: Uso de CPU e memória RAM durante o treino de *FastText* com vetores de tamanho igual a 128 ou 512.

e CPU. O aumento do tamanho do vetor não influenciou os recursos utilizados e tempo de treino do modelo. O que se pode verificar que *Word2Vec* é realmente um modelo muito leve, que consegue treinar de forma rápida.

Por fim, treinou-se o modelo *FastText*, que como se pode observar nas figuras em 4.5, comparativamente aos recursos utilizados pelo modelo *Word2Vec* são gráficos diferentes. O *FastText* utiliza CPU da mesma forma que o *Word2Vec*, no entanto usa muito mais memória RAM dependendo do tamanho do vetor, o que faz sentido pelo facto de que este tipo de modelo trata as palavras como *n-grams*, ou seja, para a mesma palavra, será gerado várias sub-palavras, o que faz com que haja mais dados para se guardar em memória. Quanto maior os vetores, como expectável, o uso de memória é maior, tendo um aumento aproximadamente de 10%. Para além disso, o modelo demora 14 horas na fase de treino, independentemente do tamanho dos vetores, o que se pode confirmar com o estudo feito previamente, que é um modelo que é muito rápido na fase de treino.

Depois de ter sido feito esta análise sobre os recursos utilizados, é importante verificar

como os modelos nativos de *word embedding* comportam-se a medir similaridade entre termos do vocabulário. No estudo feito anteriormente, muitos trabalhos, que utilizam este tipo de modelos, usam a similaridade do cosseno entre vetores para medir o quanto uma palavra é similar a outra. Como se pode ver na fórmula 4.1, a similaridade é calculada a partir do produto escalar entre dois vetores a dividir pela norma dos mesmos, sendo que o valor da similaridade varia entre -1 e 1, pelo facto que os vetores caso estejam em direções diferentes, o valor é negativo.

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} \tag{4.1}$$

A similaridade do cosseno tem a mesma escala que a similaridade calculada a partir da base de dados gerada a partir das respostas do *captcha*, o que torna mais fácil avaliar os modelos. Portanto, para cada instância dos dados de teste calculou-se a similaridade do cosseno entre os pares de vetores gerados pelos modelos de *word embedding* e comparou-se com a similaridade da base de dados. Com isto, é possível medir o MAE e MSE.

Na figura 4.6, encontra-se o gráfico relativo à avaliação feita, mas antes é importante perceber o eixo horizontal, no qual tem os nomes dos vários modelos treinados e suas especificações, ou seja, tem-se primeiramente o nome do modelo, que pode ser *FastText* seguido pelo tamanho dos vetores ou *Word2Vec* seguido pela sua arquitetura (CBOW ou *skip-gram*), algoritmo de otimização HS ou NS se tiver e, por fim, o tamanho dos vetores. Tendo isto em conta, o *FastText* é o modelo que apresentou ter valores de erro mais baixos, o que significa que conseguiu obter valores de similaridade entre pares de palavras mais próximos dos corretos. No gráfico, pode-se ver que *FastText* com vetores de tamanho 256 e 384 obtiveram um erro de MSE próximo de 0.92 e MAE de 0.93. A seguir, o modelo que teve melhores resultados foi o *Word2Vec* com uma arquitetura CBOW e *skip-gram* sem algoritmos de otimização, com vetores também de tamanho 256 e 384, atingindo um erro de *MSE* próximo de 0.94. Em vista disso, pode-se observar um padrão entre estes modelos e o *FastText* que ao gerarem vetores 128 ou 512, não conseguem estimar tão bem as similaridades como se fossem com os vetores de tamanho 256 e 384. Como 128 é o tamanho mais pequeno, pode não representar tão bem as palavras e o tamanho 512 pode ser demasiado extenso e acaba por afastar demasiado as palavras no espaço, por isso tiveram resultados piores. No entanto, os modelos de *Word2Vec* com uma arquitetura de *skip-gram* e algoritmos de otimização, tiveram resultados piores, com valores de erro entre 0.97 e 1.04, o que demonstra que os algoritmos afinal podem não ser uma boa escolha.

De forma geral, é importante realçar que os valores de erro são valores próximos de 1 e, tendo em conta, que os valores de similaridade é entre -1 e 1, significa que os modelos de *word embedding* estão a falhar redondamente. Por exemplo, se retirarmos uma instância da base de dados de um par de palavras que tem similaridade 1, ou seja, são fortemente semelhantes, ao calcular a similaridade do cosseno entre os vetores que representam as palavras dará valor próximo de zero, o que prevê que as palavras não estão relacionadas. Isto, demonstra que os modelos nativos não conseguem corretamente encontrar pares de palavras semelhantes relativamente a um problema específico de redes.

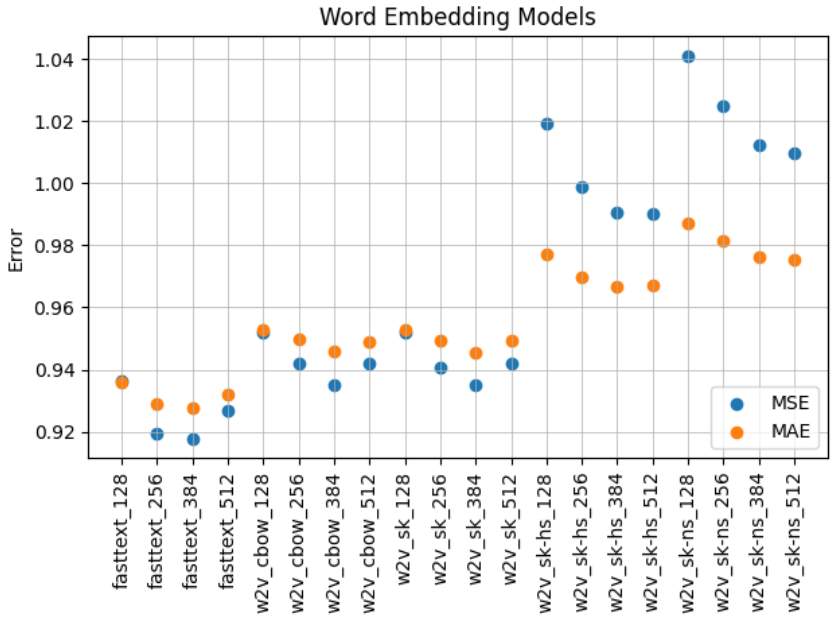


Figura 4.6: Avaliação dos modelos de *word embedding*.

4.3 Solução proposta

Para tentar melhorar os modelos nativos de *word embedding*, irá-se verificar nesta secção como a solução de adição de modelos de ML consegue melhorar a performance dos modelos. Então, após o treino dos vários modelos de *word embedding*, para cada um criou-se um ficheiro que contém os dados para treinar a rede neural, a CNN e a LSTM, ou seja, todos os ficheiros contêm os pares de palavras encontrados através do *captcha* em forma de vetor e a sua respetiva similaridade. No entanto, a representação das palavras são diferentes entre ficheiros, depende do modelo de *word embedding* que foi utilizado, pois os vetores têm valores diferentes. Portanto, antes utilizou-se a função da similaridade do cosseno para medir a semelhança entre palavras, neste caso os modelos de ML irão receber como dados de entrada os vetores dos pares de palavras, que as redes devem ajustar os seus pesos de forma a que consigam medir a similaridade entre os mesmos, retornando um valor de saída o mais próximo possível do valor de similaridade calculada a partir do *captcha*. Anteriormente avaliou-se os modelos nativos de *word embedding*, o que demonstraram que para palavras de vocabulário específico de redes e de *software* não conseguiam definir corretamente se duas palavras são ou não semelhantes. Nesta secção, irá-se analisar se os modelos de ML conjuntamente com os modelos de *word embedding* conseguem obter melhores resultados.

Como existem muitos hiperparâmetros e configurações que precisam de ser otimizados de modo a que os modelos consigam obter uma melhor performance, aplicou-se o *Grid Search* para que se consiga verificar qual as melhores combinações. Esta técnica treina os modelos para cada combinação, que são avaliados por meio de *cross-validation* e o que obtiver melhor resultado é considerado o melhor modelo. O *Grid Seatch* usa o *K-Fold cross-validation* que consiste em dividir os dados em K partes. Por exemplo, se o K for igual a três, o modelo será

Número de nós	128, 256, 384
Funções de Ativação	ReLU, Swish
Learning Rate	0.0001, 0.00001
Dropout Rate	0.2, 0.3
Otimizadores	Adam, SGD, RMSprop

Tabela 4.1: Variação de parâmetros nos modelos de ML.

treinado três vezes, sendo que na primeira vez, a primeira parte dos dados será para testar o modelo e as outras duas partes para treinar. Na segunda vez que o modelo será treinado, a primeira e terceira parte dos dados será usada para treinar o modelo e a segunda parte para teste. Na última vez, as duas primeiras partes será para treinar o modelo e a última para testar. No fim, será feito uma média dos resultados de teste. Este processo é feito para cada combinação e no final é possível encontrar a configuração que obteve melhor resultado.

A partir das arquiteturas mais estáveis de cada modelo de ML, aplicou-se a variação de hiperparâmetros apresentado na tabela 4.1, o que dá no total de 72 combinações. É de salientar que na última camada de todas as redes é constituída por um nó com a função de ativação sigmoid, apenas nas outras camadas que se alterou o número de nós e as suas funções de ativação. Para além disso, variou-se o *dropout rate*, associadas às camadas regularizadoras da rede, que definem a quantidade de nós que serão ignorados durante a fase de treino de forma a evitar *overfitting*. Por fim, variou-se os otimizadores e os seus respetivos *learning rate* para ver qual a melhor configuração para o modelo convergir e encontrar os melhores parâmetros internos de modo a obter o menor erro possível.

A biblioteca Scikit-learn fornece o *Grid Search* implementado, nomeadamente a classe GridSearchCV, que permite-nos definir a variação dos hiperparâmetros, o número de *K-Folds* para o *cross-validation* e saber qual a combinação que obteve melhores resultados. Para tornar este processo mais rápido, usou-se o método *Early Stopping* do Keras[2], que funciona como um *callback* com o objetivo de parar o treino dos modelos caso o erro de validação não tenha melhorado em N iterações. Então, para executar o *Grid Search*, que irá treinar os modelos para cada combinação, definiu-se que cada modelo treinasse no total 300 épocas, mas com o *callback Early Stopping*, se o erro de validação do *cross-validation* não melhorasse em 50 épocas, para o treino e guarda os pesos da rede da época que obteve melhor performance. No fim do *Grid Search*, pegou-se no melhor modelo e previu-se as similaridades entre as palavras dos dados de teste e calculou-se os coeficientes de correlação de Pearson e de Spearman. Com isto, agrupou-se os melhores resultados em tabela para se tentar perceber se há algum padrão de configuração que pode facilitar a aprendizagem dos modelos. No entanto, na apêndice pode-se consultar as tabelas A.2 com todos os resultados obtidos.

Portando, começando pela rede neural, na tabela 4.2 encontra-se as melhores combinações, onde se obteve os maiores valores de coeficiente de Pearson e Spearman. Pode-se notar inicialmente que a rede neural consegue adaptar-se bem com os vetores gerados pelo *Word2Vec* com uma arquitetura *skip-gram*, independentemente do tamanho dos vetores. De forma

[2]https://keras.io/api/callbacks/early_stopping/

geral, há alguns hiperparâmetros que se mantiveram como preferíveis, como por exemplo, o uso da função de ativação ReLU, enquanto que a função Swish foi apenas escolhida como a melhor para os vetores gerados pelo *Word2Vec* com arquitetura *skip-gram*, *negative sampling* e vetores de dimensão 256 (w2v_sk-ns_256). Nas camadas de *Dropout*, conseguiu-se melhores combinações com uma taxa de 0.3 em vez de 0.2. Em relação aos otimizadores, a rede neural é mais eficaz com o SGD ou RMSprop na maioria dos casos. O número de nós e *learning rate* não há um padrão que implica que a rede neural consiga obter uma melhor performance, por isso, nas combinações apresentadas na tabela, os valores escolhidos para ambos, no geral são sempre diferentes. Por fim, a rede neural com valores de entrada gerados pelo *Word2Vec* com arquitetura CBOW e vetores com 128 dimensões, obtiveram os melhores resultados de coeficientes de Pearson e Spearman, com valores 0.95 e 0.90 respetivamente. O que significa uma relação positiva muito forte entre as similaridades dos dados de teste com as similaridades previstas pelo modelo.

Modelos de Word Embedding	**Coeficiente de Pearson**	**Coeficiente de Spearman**	**Nº de nós**	***Dropout Rate***	**Função de Ativação**	**Learning Rate**	**Otimizador**
fasttext_256	0.52	0.53	256	0.3	Relu	0.00001	SGD
w2v_cbow_128	**0.95**	**0.90**	**384**	**0.2**	**Relu**	**0.00001**	**RMSprop**
w2v_sk-ns_256	0.80	0.75	256	0.3	Swish	0.0001	RMSprop
w2v_sk_128	0.59	0.52	384	0.3	Relu	0.0001	SGD
w2v_sk_256	0.88	0.84	128	0.3	Relu	0.00001	RMSprop
w2v_sk_384	0.71	0.63	256	0.2	Relu	0.00001	SGD
w2v_sk_512	0.90	0.86	128	0.3	Relu	0.0001	Adam

Tabela 4.2: Combinações de hiperparâmetros na rede neural que obtiveram valores de coeficientes de correlação maiores.

Se observarmos a tabela 4.3, que é relativa às combinações de hiperparâmetros no modelo CNN que tiveram os melhores resultados, tem muito menos casos do que a rede neural simples e os valores de coeficientes de correlação são valores piores. O melhor resultado de coeficiente de Pearson e Spearman foi 0.22 e 0.23 respetivamente, o que significa que não há relação entre os conjuntos de similaridades previstas e corretas. Como a CNN é uma rede profunda muito mais complexa que uma rede neural simples, poderia ser necessário muitos mais dados para treinar este tipo de modelo. A base de dados é muito limitada, o que pode ter prejudicado a CNN por não haver dados suficientes para encontrar padrões e conseguir prever corretamente as similaridades entre os termos.

Modelos de Word Embedding	**Coeficiente de Pearson**	**Coeficiente de Spearman**	**Nº de nós**	***Dropout Rate***	**Função de Ativação**	**Learning Rate**	**Otimizador**
fasttext_256	0.12	0.03	384	0.3	Swish	0.00001	Adam
w2v_sk-hs_128	0.12	0.10	128	0.2	Relu	0.0001	SGD
w2v_sk_128	**0.22**	**0.23**	**256**	**0.2**	**Relu**	**0.0001**	**SGD**

Tabela 4.3: Combinações de hiperparâmetros na CNN que obtiveram valores de coeficientes de correlação maiores.

Por fim, tem-se a tabela 4.4 com os resultados da rede LSTM, que como se pode observar teve também poucas combinações com coeficientes de correlação positivos. No entanto, ao contrário da CNN teve uma combinação que conseguiu obter valores altos de correlação. Na

primeira linha da tabela, pode-se ver que a LSTM com vetores de 384 dimensões gerados pelo *Word2Vec* com uma arquitetura CBOW, conseguiu obter 0.62 de coeficiente de Pearson e 0.55 de Spearman, o que significa uma relação positiva forte entre os conjuntos de similaridades. Para este caso, o modelo usou ReLU como função de ativação e o otimizador RMSprop. Estas duas configurações também foram muita vez escolhidas na rede neural, sendo uma combinação que demonstra ter algum potencial a tornar as redes mais aptas na sua aprendizagem.

Modelos de Word Embedding	**Coeficiente de Pearson**	**Coeficiente de Spearman**	**Nº de nós**	***Dropout Rate***	**Função de Ativação**	**Learning Rate**	**Otimizador**
w2v_cbow_384	**0.62**	**0.55**	**128**	**0.2**	**Relu**	**0.00001**	**RMSprop**
w2v_cbow_512	0.14	0.12	256	0.3	Relu	0.00001	SGD
w2v_sk-ns_128	0.19	0.20	256	0.3	Swish	0.0001	Adam

Tabela 4.4: Combinações de hiperparâmetros na LSTM que obtiveram valores de coeficientes de correlação maiores.

Para finalizar a avaliação, calculou-se os coeficientes de correlação para avaliar os modelos de *word embedding* de forma a comparar com os modelos de ML. Como se tinha medido as similaridades entre os vetores dos termos dos dados de teste através da distância do cosseno, calculou-se os coeficientes de Pearson e Spearman com as similaridades reais. Deste modo, construiu-se o gráfico 4.7 que demonstra os resultados dos modelos de *word embedding* e da solução proposta. Na figura 4.6, tinha-se visto que o *FastText* foi o *word embedding* que conseguiu obter menor erro comparativamente aos outros modelos, portanto, faz sentido que também seja o modelo que tenha consigo valores de correlação maiores. O *FastText* tem valores de coeficiente de Pearson próximos de 0.5, que demonstra uma relação positiva moderada entre as similaridades, mas de Spearman tem valores abaixo de 0.3, sendo uma relação fraca. Por conseguinte, não há uma consistência nos coeficientes de correlação. Os restantes modelos tiveram valores próximos de zero, o que se confirma a avaliação feita anteriormente, que os modelos nativos de *word embedding* não são suficientes para se conseguir medir similaridade entre palavras de vocabulário de redes e *software*. No lado direito do gráfico, encontra-se os resultados obtidos pela solução proposta, que estão sumariados em formato tabela na apêndice A.2. Pode-se observar que há valores de correlação igual a zero, como por exemplo, valores associados ao fasttext_128, isto porque os modelos de ML divergiram em vez de convergir, o que fez com que não conseguissem retornar valores de previsão de similaridade durante a fase de testes. Este problema pode ocorrer pelo facto das arquiteturas dos modelos de ML ou as configurações dos hiperparâmetros, não serem o ideal para os modelos se adaptarem aos dados, mas também pode ser devido à base de dados ser limitada. Por isso, como a rede neural é uma rede muito mais simples do que a CNN e LSTM, foi a que conseguiu obter melhores resultados, principalmente combinada com os vetores gerados pelo *Word2Vec* com uma arquitetura *skip-gram*. As *features* geradas pelo modelo *Word2Vec* com uma arquitetura *skip-gram* e com algoritmo de otimização HS, independentemente da dimensão dos vetores, com a adição dos modelos de ML, obteve-se valores de correlação próximos de zero, sendo uma combinação pouco efetiva. Embora a CNN e LSTM não tenham valores tão bons como a rede neural, com uma base de dados de maior dimensões, podem ser capazes de fazer esta

análise de semelhança semântica de forma eficaz.

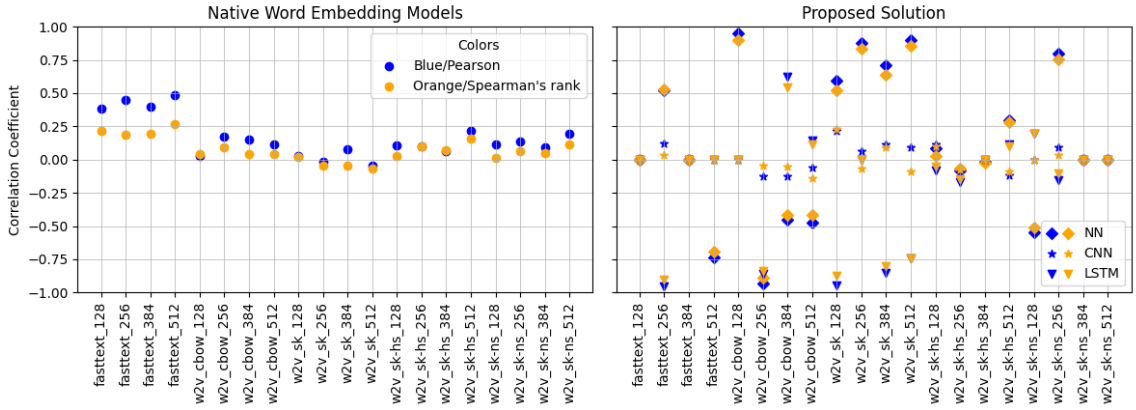


Figura 4.7: Gráfico relativo à avaliação realizada aos modelos *word embedding* e de ML.

4.4 Sumário

Neste capítulo descreve essencialmente as avaliações realizadas aos modelos de *word embedding* e de ML, tendo como base as similaridades medidas através das respostas obtidas pelo *captcha*.

Na secção 4.1, mostra que as respostas obtidas, que na maioria dos casos para um determinado par de termos só foi votado uma vez, o que torna o par de semelhança pouco confiável. Por isso, recorreu-se a técnicos da área para ajustar as similaridades. Para além disso, dividiu-se os dados em duas partes, uma para treinar os modelos de ML e outra para testar. Escolheu-se uma divisão 90% e 10% respetivamente, dando mais importância ao número de dados de treino, pelo facto de a base de dados não ter um tamanho considerável. Os 10% de dados de teste, foram escolhidos a partir dos pares mais votados no *captcha*.

Durante o treino dos modelos de *word embedding*, mediu-se os recursos utilizados, como o uso de CPU e de memória RAM, e confirmou-se que são modelos leves. Na secção 4.2, para avaliar este tipo de modelos, como têm definidos vetores para cada palavra do vocabulário, para cada par de palavra pertencentes aos dados de teste, mediu-se a similaridade do cosseno. Com isto, calculou-se o MSE e MAE entre as similaridades do cosseno com as esperadas. Cada modelo de *word embedding* obteve valores de erro muito próximos de 1, o que demonstra que falhavam redondamente, ou seja, as similaridades previstas estavam muito longe das reais.

De seguida, apresenta-se as experiências aplicadas à solução proposta proposta na secção 4.3 e os seus respetivos resultados. Aplicou-se o *Grid Searh* com intuito de criar diferentes combinações de hiperparâmetros e saber qual a melhor configuração para cada rede. No final desta secção é possível visualizar a diferença entre os modelos de *word embedding* com a solução proposta. Efetivamente, os modelos de ML demonstram ter potencial para calcular semelhança entre palavras relacionadas com vocabulário de redes e *software*.

CAPÍTULO 5

Conclusão e trabalho futuro

5.1 Considerações finais

Este projeto começou com o tratamento de uma base de dados com um tamanho significativo de diagnósticos de falhas relativos a dispositivos reais de redes. Todo este processo era importante para remover o máximo ruído dos dados e definir o vocabulário de domínio específico de redes e *software*.

Desta forma, treinou-se os modelos nativos de *word embedding* de modo a representarem cada palavra do vocabulário em vetores numéricos. Na avaliação feita a estes modelos, foi possível verificar que falhavam redondamente na previsão das similaridades, demonstrando assim que não são capazes de medir a similaridade apenas com a distância do cosseno entre os vetores. No entanto, o *FastText* foi o modelo que teve melhores resultados.

Com a adição dos modelos de ML, que utilizam os vetores gerados pelos modelos de *word embedding*, a rede neural simples combinado principalmente com vetores do *Word2Vec* com uma arquitetura de *skip-gram* obteve coeficientes de correlação próximos de 1, sendo a rede que teve melhor performance. Na experiência realizada com o *Grid Search*, onde se aplicou diferentes combinações de hiperparâmetros, na rede neural foi de se notar que a função de ativação ReLU era maioritariamente escolhida, embora que a função de ativação *Swish* foi provada em várias experiências que era melhor que a função ReLU. Em relação aos otimizadores, as funções RMSprop e SGD foram configurações que demonstraram influenciar a aprendizagem da rede neural, tornando-as mais aptas para a resolução do problema. A CNN e LSTM são redes muito mais complexas e como a quantidade de dados para treinar estes modelos não era demasiado grande, era difícil fazer com que estas redes conseguissem encontrar um padrão ou criar uma função de similaridade a partir dos dados de treino. Contudo, no estudo feito antes de desenvolver o projeto, são modelos que têm uma grande capacidade para resolução de problemas deste tipo, portanto não devem ser descartados.

Em suma, nesta dissertação mostrou-se que os modelos nativos de *word embedding* para medir a similaridade entre as palavras relacionadas com um vocabulário específico de redes e *software*, não são suficientes. Contudo, a combinação das *features* geradas por estes modelos

com as redes neurais, apresentam ter um forte potencial para a resolução do problema. Embora a base de dados tenha sido limitada, a solução proposta demonstrou ter a capacidade de encontrar uma função eficiente para calcular a similaridade entre as representações vetoriais das palavras. Para além disso, a solução está implementada de forma modular, o que no caso de futuramente aparecer novos modelos seja fácil de substituir pelos modelos do nosso sistema e seja aplicado testes exaustivos de modo a encontrar a melhor configuração para a resolução do problema.

5.2 Trabalho futuro

Nesta secção, será apresentado o trabalho que falta ser feito e, caso este trabalho continue, que direções pode seguir.

Um dos maiores problemas do projeto é a limitação dos dados para treinar os modelos de ML, por isso, aumentar a quantidade de dados é um passo importante que pode melhorar significativamente a performance das redes, tais como, CNN e LSTM. Ainda, faria com que se garantisse maior confiabilidade dos pares criados e aumentasse a cobertura do vocabulário.

O mapeamento dos diagnósticos de alarmes para um modelo único foi um objectivo que não foi realizado devido ao atraso causado pela recolha de respostas do *captcha* de palavras. Seria um algoritmo que teria como base os modelos desenvolvidos nesta dissertação, com o objetivo de verificar se dois diagnósticos de falhas estão no âmbito do mesmo contexto ou não e mapeá-los para a sua respetiva falha.

Com isto, é um modelo que pode ter um impacto na automação das redes, porque caso se consiga mapear diferentes diagnósticos para a mesma falha, seria interessante ter outro módulo que tenha a capacidade de fazer outro mapeamento das falhas para a sua respetiva solução, caso seja possível. Desta forma, este processo para redes complexas, se for suficientemente robusto, terá muita importância na reparação de falhas e a tornar as redes num sistema de *self-healing*.

Referências

[1] H. Mfula e J. K. Nurminen, «Adaptive Root Cause Analysis for Self-Healing in 5G Networks,» em *2017 International Conference on High Performance Computing Simulation (HPCS)*, 2017, pp. 136–143. DOI: 10.1109/HPCS.2017.31.

[2] P. B. Seokar, L. Nayak e P. Patil, «Intelligent Adapter for Simplified Central Monitoring of Network Alarms,» em *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2020, pp. 1–6. DOI: 10.1109/CONECCT50063.2020.9198467.

[3] O. S. Larry Peterson, «5G Mobile Networks: A Systems Approach,» Open Networking Foundation, 2020. URL: https://5g.systemsapproach.org/intro.html.

[4] G. Mustafa Zebari, D. Assad Zebari e A. Al-zebari, «FUNDAMENTALS OF 5G CELLULAR NETWORKS: A REVIEW,» *Journal of Information Technology and Informatics*, vol. 1, n.º 1, pp. 1–5, abr. de 2021. DOI: 10.6084. URL: https://qabasjournals.com/index.php/jiti/article/view/22.

[5] T. Fisher, «5G: Everything You Need to Know,» Lifewire, 2022. URL: https://www.lifewire.com/5g-wireless-4155905.

[6] W. Khan, A. Daud, J. A. Nasir e T. Amjad, «A survey on the state-of-the-art machine learning models in the context of NLP,» *kuwait journal of science*, vol. 43, 2016.

[7] D. Khyani e S. B S, «An Interpretation of Lemmatization and Stemming in Natural Language Processing,» *Shanghai Ligong Daxue Xuebao/Journal of University of Shanghai for Science and Technology*, vol. 22, pp. 350–357, jan. de 2021.

[8] Y. Wu, J. C. Denny, S. Trent Rosenbloom et al., «A long journey to short abbreviations: developing an open-source framework for clinical abbreviation recognition and disambiguation (CARD),» *Journal of the American Medical Informatics Association*, vol. 24, n.º e1, e79–e86, ago. de 2016, ISSN: 1067-5027. DOI: 10.1093/jamia/ocw109. URL: https://doi.org/10.1093/jamia/ocw109.

[9] Y. Wu, S. T. Rosenbloom, J. C. Denny et al., «Detecting abbreviations in discharge summaries using machine learning methods.,» *AMIA ... Annual Symposium proceedings. AMIA Symposium*, vol. 2011, pp. 1541–9, 2011.

[10] C.-J. Kuo, M. H. T. Ling, K.-T. Lin e C.-N. Hsu, «BIOADI: a machine learning approach to identifying abbreviations and definitions in biological literature,» *BMC Bioinformatics*, vol. 10, S7–S7, 2009. DOI: https://doi.org/10.1186/1471-2105-10-S15-S7.

[11] L. Andersson, M. Lupu, J. Palotti, A. Hanbury e A. Rauber, «When is the Time Ripe for Natural Language Processing for Patent Passage Retrieval?» Em *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, sér. CIKM '16, Indianapolis, Indiana, USA: Association for Computing Machinery, 2016, pp. 1453–1462. DOI: 10.1145/2983323.2983858.

[12] R. Boos, K. Prestes e A. Villavicencio, «Identification of Multiword Expressions in the brWaC,» em *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland: European Language Resources Association (ELRA), mai. de 2014, pp. 728–735. URL: http://www.lrec-conf.org/proceedings/lrec2014/pdf/518_Paper.pdf.

[13] L. Huang e C. Ling, «Representing Multiword Chemical Terms through Phrase-Level Preprocessing and Word Embedding,» *ACS Omega*, vol. 4, n.º 20, pp. 18 510–18 519, 2019, PMID: 31737809. DOI: 10.1021/acsomega.9b02060. URL: https://doi.org/10.1021/acsomega.9b02060.

[14] M. Atzori e S. Balloccu, «Fully-Unsupervised Embeddings-Based Hypernym Discovery,» *Information*, vol. 11, n.º 5, 2020, ISSN: 2078-2489. DOI: 10.3390/info11050268.

[15] M. A. Hearst, «Automatic Acquisition of Hyponyms from Large Text Corpora,» sér. COLING '92, Nantes, France: Association for Computational Linguistics, 1992, pp. 539–545. DOI: 10.3115/992133.992154.

[16] D. Lin, S. Zhao, L. Qin e M. Zhou, «Identifying Synonyms among Distributionally Similar Words,» em *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, sér. IJCAI'03, Acapulco, Mexico: Morgan Kaufmann Publishers Inc., 2003, pp. 1492–1493.

[17] E. Westerhout e P. Monachesi, «Extraction of Dutch definitory contexts for eLearning purposes,» *Lot Occasional Series*, vol. 7, pp. 219–234, 2007.

[18] J. Seitner, C. Bizer, K. Eckert et al., «A Large DataBase of Hypernymy Relations Extracted from the Web.,» em *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, European Language Resources Association (ELRA), mai. de 2016, pp. 360–367.

[19] A. Ly, B. Uthayasooriyar e T. Wang, *A survey on natural language processing (nlp) and applications in insurance*, 2020. DOI: 10.48550/ARXIV.2010.00462.

[20] L. Zhang, J. Li e C. Wang, «Automatic synonym extraction using Word2Vec and spectral clustering,» em *2017 36th Chinese Control Conference (CCC)*, 2017, pp. 5629–5632. DOI: 10.23919/ChiCC.2017.8028251.

[21] E. Levi, S. Herman e A. Rappoport, *Computing Word Classes Using Spectral Clustering*, 2018. DOI: 10.48550/ARXIV.1808.05374.

[22] S. Larabi Marie-Sainte, N. Alalyani, S. Alotaibi, S. Ghouzali e I. Abunadi, «Arabic Natural Language Processing and Machine Learning-Based Systems,» *IEEE Access*, vol. 7, pp. 7011–7020, 2019. DOI: 10.1109/ACCESS.2018.2890076.

[23] S.-C. Wang, «Artificial Neural Network,» em *Interdisciplinary Computing in Java Programming*. Boston, MA: Springer US, 2003, pp. 81–100, ISBN: 978-1-4615-0377-4. DOI: 10.1007/978-1-4615-0377-4_5. URL: https://doi.org/10.1007/978-1-4615-0377-4_5.

[24] P. Ramachandran, B. Zoph e Q. V. Le, «Searching for Activation Functions,» *ArXiv*, 2018. DOI: 10.48550/ARXIV.1710.05941. URL: https://arxiv.org/abs/1710.05941.

[25] M. Hardt, B. Recht e Y. Singer, «Train Faster, Generalize Better: Stability of Stochastic Gradient Descent,» em *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, sér. ICML'16, New York, NY, USA: JMLR.org, 2016, pp. 1225–1234. DOI: 10.48550/ARXIV.1509.01240. URL: https://arxiv.org/abs/1509.01240.

[26] D. Kingma e J. Ba, «Adam: A Method for Stochastic Optimization,» *International Conference on Learning Representations*, dez. de 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[27] Y. Tan, X. Wang e T. Jia, «From Syntactic Structure to Semantic Relationship: Hypernym Extraction from Definitions by Recurrent Neural Networks Using the Part of Speech Information,» em nov. de 2020, pp. 529–546. DOI: 10.1007/978-3-030-62419-4_30.

[28] X. Zhang, M. H. Chen e Y. Qin, «NLP-QA Framework Based on LSTM-RNN,» em *2018 2nd International Conference on Data Science and Business Analytics (ICDSBA)*, 2018, pp. 307–311. DOI: 10.1109/ICDSBA.2018.00065.

[29] A. Graves, A.-r. Mohamed e G. Hinton, «Speech recognition with deep recurrent neural networks,» em *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 6645–6649. DOI: 10.1109/ICASSP.2013.6638947.

[30] T. Nguyen e R. Grishman, «Relation Extraction: Perspective from Convolutional Neural Networks,» jan. de 2015, pp. 39–48. DOI: 10.3115/v1/W15-1506.

[31] G. A. Miller, «WordNet: A Lexical Database for English,» *Commun. ACM*, vol. 38, n.º 11, pp. 39–41, nov. de 1995. DOI: 10.1145/219717.219748.

[32] H. Liu e P. Singh, «ConceptNet—A Practical Commonsense Reasoning Tool-Kit,» *BT technology journal*, vol. 22, jun. de 2004. DOI: 10.1023/B:BTTJ.0000047600.45421.6d.

[33] F. Shi, L. Chen, J. Han e P. Childs, «A Data-Driven Text Mining and Semantic Network Analysis for Design Information Retrieval,» *Journal of Mechanical Design*, vol. 139, n.º 11, out. de 2017. DOI: 10.1115/1.4037649.

[34] S. Sarica, J. Luo e K. L. Wood, «TechNet: Technology semantic network based on patent data,» *Expert Systems with Applications*, vol. 142, p. 112 995, 2020. DOI: https://doi.org/10.1016/j.eswa.2019.112995. URL: https://www.sciencedirect.com/science/article/pii/S0957417419307122.

[35] Z. S. Harris, «Distributional Structure,» *WORD*, vol. 10, n.º 2-3, pp. 146–162, 1954. DOI: 10.1080/00437956.1954.11659520. eprint: https://doi.org/10.1080/00437956.1954.11659520. URL: https://doi.org/10.1080/00437956.1954.11659520.

[36] S. Mohammad, «Measuring Semantic Distance Using Distributional Profiles of Concepts,» tese de doutoramento, CAN, 2008.

[37] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai e X. Huang, «Pre-trained models for natural language processing: A survey,» *Science China Technological Sciences*, vol. 63, pp. 1872–1897, out. de 2020. DOI: 10.1007/s11431-020-1647-3.

[38] T. Mikolov, K. Chen, G. Corrado e J. Dean, *Efficient Estimation of Word Representations in Vector Space*, 2013. arXiv: 1301.3781 [cs.CL].

[39] T. Mikolov, I. Sutskever, K. Chen, G. Corrado e J. Dean, *Distributed Representations of Words and Phrases and their Compositionality*, 2013. arXiv: 1310.4546 [cs.CL].

[40] M. Naili, A. H. Chaibi e H. H. Ben Ghezala, «Comparative study of word embedding methods in topic segmentation,» *Procedia Computer Science*, vol. 112, pp. 340–349, 2017, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.08.009.

[41] J. Pennington, R. Socher e C. Manning, «GloVe: Global Vectors for Word Representation,» em *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, out. de 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162.

[42] O. Levy, Y. Goldberg e I. Dagan, «Improving Distributional Similarity with Lessons Learned from Word Embeddings,» *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 211–225, 2015. DOI: 10.1162/tacl_a_00134.

[43] P. Bojanowski, E. Grave, A. Joulin e T. Mikolov, «Enriching Word Vectors with Subword Information,» *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, jun. de 2017. DOI: 10.1162/tacl_a_00051.

[44] M. E. Peters, M. Neumann, M. Iyyer et al., «Deep Contextualized Word Representations,» em *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, Association for Computational Linguistics, jun. de 2018, pp. 2227–2237. DOI: 10.18653/v1/N18-1202.

[45] L. Zhuang, L. Wayne, S. Ya e Z. Jun, «A Robustly Optimized BERT Pre-training Approach with Post-training,» em *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, Chinese Information Processing Society of China, ago. de 2021, pp. 1218–1227. URL: https://aclanthology.org/2021.ccl-1.108.

[46] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma e R. Soricut, «ALBERT: A Lite BERT for Self-supervised Learning of Language Representations,» em *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, OpenReview.net, 2020. URL: https://openreview.net/forum?id=H1eA7AEtvS.

[47] R. N. Al-Matham, H. S. Al-Khalifa e M. I. Uddin, «SynoExtractor: A Novel Pipeline for Arabic Synonym Extraction Using Word2Vec Word Embeddings,» *Complex.*, vol. 2021, jan. de 2021, ISSN: 1076-2787. DOI: 10.1155/2021/6627434.

[48] V. Shwartz, Y. Goldberg e I. Dagan, «Improving Hypernymy Detection with an Integrated Path-based and Distributional Method,» em *Proceedings of the 54th Annual Meeting of the Association for*

Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, ago. de 2016, pp. 2389–2398. DOI: 10.18653/v1/P16-1226.

[49] Y. Kim, «Convolutional Neural Networks for Sentence Classification,» em *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar: Association for Computational Linguistics, out. de 2014, pp. 1746–1751. DOI: 10.3115/v1/D14-1181. URL: https://aclanthology.org/D14-1181.

[50] H. Choi, J. Kim, S. Joe e Y. Gwon, «Evaluation of BERT and ALBERT Sentence Embedding Performance on Downstream NLP Tasks,» em *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021, pp. 5482–5487. DOI: 10.1109/ICPR48806.2021.9412102.

[51] L. Tóth, B. Nagy, T. Gyimóthy e L. Vidács, «Mining Hypernyms Semantic Relations from Stack Overflow,» em *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, sér. ICSEW'20, Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 360–366. DOI: 10.1145/3387940.3392160. URL: https://doi.org/10.1145/3387940.3392160.

[52] X. Chen, C. Chen, D. Zhang e Z. Xing, «SEthesaurus: WordNet in Software Engineering,» *IEEE Transactions on Software Engineering*, vol. 47, n.º 9, pp. 1960–1979, 2021. DOI: 10.1109/TSE.2019.2940439.

[53] G. Van Rossum e F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697.

[54] C. R. Harris, K. J. Millman, S. J. van der Walt et al., «Array programming with NumPy,» *Nature*, vol. 585, n.º 7825, pp. 357–362, set. de 2020. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[55] E. Loper e S. Bird, «NLTK: the natural language toolkit,» em *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, 2006, pp. 69–72. DOI: 10.48550/ARXIV.CS/0205028. URL: https://arxiv.org/abs/cs/0205028.

[56] F. Pedregosa, G. Varoquaux, A. Gramfort et al., «Scikit-learn: Machine Learning in Python,» *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. DOI: 10.48550/ARXIV.1201.0490.

[57] M. Abadi, A. Agarwal, P. Barham et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, 2016. DOI: 10.48550/ARXIV.1603.04467. URL: https://arxiv.org/abs/1603.04467.

[58] P. Barrett, J. Hunter, J. T. Miller, J. .-. Hsu e P. Greenfield, «matplotlib – A Portable Python Plotting Package,» em *Astronomical Data Analysis Software and Systems XIV*, P. Shopbell, M. Britton e R. Ebert, eds., sér. Astronomical Society of the Pacific Conference Series, vol. 347, dez. de 2005, p. 91.

[59] V. Satopaa, J. Albrecht, D. Irwin e B. Raghavan, «Finding a "Kneedle"in a Haystack: Detecting Knee Points in System Behavior,» em *2011 31st International Conference on Distributed Computing Systems Workshops*, 2011, pp. 166–171. DOI: 10.1109/ICDCSW.2011.20.

[60] R. Dunford, Q. Su e E. Tamang, «The Pareto Principle,» *The Plymouth Student Scientist*, vol. 7, pp. 140–148, 2014.

[61] S. Salvador e P. Chan, «Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms,» em *16th IEEE International Conference on Tools with Artificial Intelligence*, 2004, pp. 576–584. DOI: 10.1109/ICTAI.2004.50.

[62] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado e J. Dean, «Distributed Representations of Words and Phrases and their Compositionality,» em *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani e K. Weinberger, eds., vol. 26, Curran Associates, Inc., 2013. DOI: 10.48550/ARXIV.1310.4546. URL: https://arxiv.org/abs/1310.4546.

[63] A. F. Agarap, «Deep Learning using Rectified Linear Units (ReLU),» *CoRR*, vol. abs/1803.08375, 2018. arXiv: 1803.08375. URL: http://arxiv.org/abs/1803.08375.

APÊNDICE A

A.1 Uso de recursos dos modelos de *word embedding*

Modelos de Word Embedding	**Cpu (%)**	**Memória Ram (%)**	**Tempo (s)**
fasttext_128	4.29	3.34	49915
fasttext_256	4.29	6.20	50260
fasttext_384	4.28	9.11	51010
fasttext_512	4.29	11.98	50740
w2v_cbow_128	4.29	0.51	50150
w2v_cbow_256	4.29	0.58	50550
w2v_cbow_384	4.29	0.64	50795
w2v_cbow_512	4.30	0.68	49355
w2v_sk_128	4.34	0.49	49880
w2v_sk_256	4.33	0.57	49020
w2v_sk_384	4.33	0.64	49715
w2v_sk_512	4.33	0.70	50450
w2v_sk-hs_128	64.24	0.58	90020
w2v_sk-hs_256	73.20	0.76	117490
w2v_sk-hs_384	79.78	0.86	143640
w2v_sk-hs_512	86.79	1.02	173620
w2v_sk-ns_128	42.14	0.56	55585
w2v_sk-ns_256	48.83	0.65	28035
w2v_sk-ns_384	62.40	0.83	86060
w2v_sk-ns_512	67.90	0.97	100390

Tabela A.1: Recursos utilizados para treinar os modelos de *word embedding*, tais como, o tempo total da fase de treino, a média do uso de CPU e da memória RAM.

A.2 Resultados da solução proposta

Modelos de Word Embedding	Coeficiente de Pearson	Coeficiente de Spearman	Nº de nós	*Dropout Rate*	Função de Ativação	Learning Rate	Otimizador
fasttext_256	0.52	0.53	256	0.3	Relu	0.00001	SGD
fasttetxt_384	0.00	0.00	384	0.3	Swish	0.0001	SGD
fasttext_512	-0.73	-0.69	128	0.2	Swish	0.00001	RMSprop
w2v_cbow_128	**0.95**	**0.90**	**384**	**0.2**	**Relu**	**0.00001**	**RMSprop**
w2v_cbow_256	-0.93	-0.89	256	0.3	Relu	0.00001	Adam
w2v_cbow_384	-0.45	-0.42	256	0.2	Relu	0.0001	SGD
w2v_cbow_512	-0.47	-0.42	384	0.3	Relu	0.0001	SGD
w2v_sk-hs_128	0.08	0.02	256	0.3	Relu	0.00001	SGD
w2v_sk-hs_256	-0.08	-0.07	256	0.3	Relu	0.00001	SGD
w2v_sk-hs_384	-0.02	-0.03	128	0.3	Relu	0.0001	SGD
w2v_sk-hs_512	0.29	0.28	256	0.3	Relu	0.0001	SGD
w2v_sk-ns_128	-0.55	-0.51	384	0.3	Relu	0.0001	SGD
w2v_sk-ns_256	0.80	0.75	256	0.3	Swish	0.0001	RMSprop
w2v_sk_128	0.59	0.52	384	0.3	Relu	0.0001	SGD
w2v_sk_256	0.88	0.84	128	0.3	Relu	0.00001	RMSprop
w2v_sk_384	0.71	0.63	256	0.2	Relu	0.00001	SGD
w2v_sk_512	0.90	0.86	128	0.3	Relu	0.0001	Adam

Tabela A.2: Coeficientes de Pearson e de Spearman de todas as combinações de hiperparâmetros na rede neural.

Modelos de Word Embedding	Coeficiente de Pearson	Coeficiente de Spearman	Nº de nós	*Dropout Rate*	Função de Ativação	Learning Rate	Otimizador
fasttext_256	0.12	0.03	384	0.3	Swish	0.00001	Adam
w2v_cbow_256	-0.12	-0.05	128	0.2	Swish	0.00001	SGD
w2v_cbow_384	-0.12	-0.06	128	0.2	Swish	0.00001	RMSprop
w2v_cbow_512	-0.06	-0.14	256	0.2	Swish	0.00001	SGD
w2v_sk-hs_128	0.12	0.10	128	0.2	Relu	0.0001	SGD
w2v_sk-hs_256	-0.10	-0.08	128	0.2	Swish	0.00001	SGD
w2v_sk-hs_512	-0.12	-0.09	384	0.2	Swish	0.00001	RMSprop
w2v_sk-ns_128	0.00	0.00	128	0.2	Swish	0.0001	RMSprop
w2v_sk-ns_256	0.09	0.04	128	0.2	Relu	0.0001	RMSprop
w2v_sk_128	**0.22**	**0.23**	**256**	**0.2**	**Relu**	**0.0001**	**SGD**
w2v_sk_256	0.06	-0.07	256	0.3	Swish	0.00001	SGD
w2v_sk_384	0.11	0.09	384	0.2	Swish	0.0001	Adam
w2v_sk_512	0.09	-0.09	128	0.3	Relu	0.0001	RMSprop

Tabela A.3: Coeficientes de Pearson e de Spearman de todas as combinações de hiperparâmetros na cnn.

Modelos de Word Embedding	Coeficiente de Pearson	Coeficiente de Spearman	Nº de nós	*Dropout Rate*	Função de Ativação	Learning Rate	Otimizador
fasttext_256	-0.95	-0.90	256	0.2	Relu	0.0001	RMSprop
w2v_cbow_256	-0.86	-0.84	128	0.2	Relu	0.0001	RMSprop
w2v_cbow_384	**0.62**	**0.55**	**128**	**0.2**	**Relu**	**0.00001**	**RMSprop**
w2v_cbow_512	0.14	0.12	256	0.3	Relu	0.00001	SGD
w2v_sk-hs_128	-0.08	-0.05	256	0.3	Swish	0.00001	Adam
w2v_sk-hs_256	-0.17	-0.15	384	0.3	Swish	0.00001	Adam
w2v_sk-hs_512	0.11	0.10	128	0.2	Swish	0.00001	SGD
w2v_sk-ns_128	0.19	0.20	256	0.3	Swish	0.0001	Adam
w2v_sk-ns_256	-0.16	-0.10	384	0.2	Swish	0.00001	Adam
w2v_sk_128	-0.95	-0.88	384	0.2	Relu	0.0001	RMSprop
w2v_sk_384	-0.85	-0.80	128	0.3	Relu	0.0001	Adam
w2v_sk_512	-0.74	-0.74	256	0.2	Relu	0.00001	RMSprop

Tabela A.4: Coeficientes de Pearson e de Spearman de todas as combinações de hiperparâmetros na LSTM.