Universidade do Minho
Departamento de Informática

Sistemas Inteligentes

Aprendizagem Profunda

Realizado por:
Emanuel Lopes Monteiro da Silva (PG53802)
João Andrade Rodrigues (PG57879)
Jorge Eduardo Quinteiro Oliveira (PG52688)
Mateus Lemos Martins (PG57890)
Bernardo Dutra Lemos (E12338)

PG53802

PG57879

PG52688

PG57890

E12338

2 de abril de 2025
Ano Letivo 2024/25

1 Introdução

Neste relatório, descrevemos o desenvolvimento de modelos de *Machine Learning* e *Deep Learning* para distinguir entre textos gerados por inteligência artificial e textos escritos por seres humanos. Abordámos a criação e o pré-processamento dos *datasets*, recorrendo a técnicas como TF-IDF para extrair representações significativas dos textos, e implementámos diversas abordagens, desde modelos de base até modelos avançados suportados por frameworks de *Deep Learning*.

O relatório está organizado em várias secções. A secção **Metodologia para Criação do Dataset** descreve os procedimentos de recolha e pré-processamento dos dados. A secção **Implementação dos Modelos** detalha as abordagens desenvolvidas com código próprio (e.g., Logistic Regression, DNN, RNN) e com frameworks (e.g., GRU, RoBERTa, BERT, LSTM), explicando as técnicas de regularização e *fine-tuning* utilizadas. A secção de **Resultados Obtidos** apresenta as métricas de avaliação de cada modelo, acompanhadas de tabelas e gráficos, e a **Conclusão** sumariza as descobertas e sugere possíveis melhorias para futuras implementações.

2 Metodologia para criação do dataset

O grupo desenvolveu três *scripts* para criar uma *pipeline* de processamento de dados, garantindo que todo o conjunto de treino seja devidamente formatado e validado antes de ser usado nos modelos.

O **clean_pipeline.py** foi concebido para corrigir *datasets* provenientes do Kaggle, ajustando problemas de formatação, codificação e removendo caracteres indesejados. Além disso, separa os dados em ficheiros distintos para *Text* (inputs) e *Labels* (outputs), preparando-os para a fase seguinte.

O **create_train_datasets.py** constitui a fase central desta *pipeline* e é responsável por criar um *dataset* de treino com base em diversas condições. Entre essas condições, incluem-se um número mínimo e máximo de palavras em cada frase, bem como o equilíbrio entre as *labels* 'AI' e 'Human', de modo a garantir uma distribuição representativa das classes. No final, é gerado um *dataset* de treino pronto para ser utilizado no treino dos modelos.

Por fim, o **google_custom_search.py** foi aplicado apenas para validar, *ipsis verbis*, se as frases efetivamente existem no Google. Através de uma pesquisa linha a linha, tornou-se possível avaliar quantas frases poderiam, potencialmente, ter sido escritas por humanos, ajudando na comparação de resultados durante as três fases de submissão dos de classificadores. Embora tenha sido útil, não faz parte direta do *pipeline* de preparação principal, servindo apenas como uma referência adicional na análise final de desempenho dos modelos.

3 Implementação dos modelos

3.1 Implementação dos modelos de raiz

3.1.1 Logistic Regression (LR)

O modelo de *Logistic Regression* foi implementado para servir como modelo base, e conta com vetorização TF-IDF dos textos e um pipeline modular para treino e avaliação. O pré-processamento inclui a limpeza dos textos e a construção de um vocabulário com base em frequência mínima. Os dados foram divididos em *training set* (70%) e *test set* (30%), reservando ainda 20% dos dados de treino para validação.

O treino foi realizado com *gradient descent*, utilizando regularização L2 para evitar overfitting. A *hyperparameter tuning* foi conduzida através de *grid search*, variando a *learning rate* (*alpha*), o coeficiente de regularização (*lambda*) e o número de iterações.

O modelo final foi treinado sobre todo o conjunto de treino com estes parâmetros. A avaliação foi realizada com base em várias métricas: *accuracy*, *balanced accuracy*, *precision*, *recall* e *F1-score*.

Tabela 1: Avaliação das métricas no *dataset* de teste *merged_inputs* - LR

Metric	**Value**
Accuracy	92.92%
Precision	95.27%
Recall	90.66%
F1-Score	92.91%
Balanced Accuracy	92.97%

3.1.2 Deep Neural Network (DNN)

Na vertente de implementação dos modelos de raiz com foco em *neural networks*, implementamos uma DNN (Deep Neural Network) utilizando apenas NumPy para os cálculos.

Após efetuarmos um pré-processamento dos dados em bruto (e.g., limpeza do texto, normalização, remoção de *stopwords*, lematização), testámos a abordagem *Bag-of-Words*, mas os resultados demonstraram limitações na capacidade de captar a relevância dos termos. O TF-IDF atribui pesos que atenuam o impacto das palavras frequentes, realçando aquelas que são mais discriminativas em cada documento. Decidimos, então, utilizar o TF-IDF, uma vez que permitiu à nossa DNN captar nuances essenciais nos textos para distinguir corretamente entre conteúdos gerados por IA e por humanos.

A arquitetura do modelo é do tipo *feedforward*, em que os dados de entrada são passados através de várias *dense layers* (fully-connected), onde

cada camada aplica uma transformação linear seguida de uma função de ativação. Nas *hidden layers*, utilizamos a função ReLU (Rectified Linear Unit) para introduzir não-linearidade, e na camada de saída aplicamos a função sigmoide, sendo uma das mais adequadas para classificação binária.

Durante o treino da rede neuronal, é utilizado o algoritmo de *backpropagation* para ajustar os pesos e *bias* da rede. Após a *forward propagation*, o erro é calculado através da função de perda, que neste caso é a *Binary Cross-Entropy*. A derivada desta função de perda é calculada e propagada em sentido inverso através da rede para atualizar os parâmetros, utilizando um otimizador baseado em *gradient descent* com *momentum*. O uso do *momentum* ajuda a suavizar as atualizações e a acelerar a convergência.

A implementação inclui ainda uma camada de *dropout* aplicada durante o treino, para reduzir o risco de *overfitting*. O *dropout* funciona desativando aleatoriamente alguns neurónios durante a *forward propagation*, para evitar que o modelo se adapte demasiado a padrões específicos do conjunto de treino, e permitir uma maior generalização.

A nossa abordagem final combinou técnicas de regularização, como o *dropout*, com a otimização dos hiperparâmetros efetuada através de *random search*, o que permitiu reduzir significativamente o tempo de treino sem comprometer em grande termo a convergência. Observámos que, durante o treino, a *loss* foi reduzida de 0.6866 para 0.0063, e as métricas de avaliação atingiram valores elevados, o que indica que a nossa DNN conseguiu captar algumas características relevantes dos dados e realizar uma classificação razoavelmente precisa e equilibrada.

Tabela 2: Avaliação das métricas no *dataset* de teste *merged_inputs* - DNN

Metric	**Value**
Accuracy	94.69%
Precision	94.81%
Recall	94.81%
F1-Score	94.81%
Balanced Accuracy	94.69%

3.1.3 Recurrent Neural Network (RNN)

Na vertente de implementação de modelos a partir do zero, desenvolvemos ainda uma *RNN (Recurrent Neural Network)*, com o objetivo de classificar textos como "*AI*" ou "*Human*". Esta abordagem recorre a *embeddings GloVe* de dimensão *50*, aplicados após pré-processamento dos textos.

O pipeline de preparação dos dados envolveu várias etapas essenciais para melhorar a qualidade dos dados de entrada. Inicialmente, aplicámos uma limpeza textual, convertendo para minúsculas e removendo pontuação. Em

seguida, eliminámos palavras irrelevantes através de uma lista de *stopwords*. Por fim, cada texto foi convertido numa sequência de vetores de *embeddings* (*GloVe 50d*), com *padding* para um comprimento fixo e *normalização* por dimensão (*zero mean* e *unit variance*).

A arquitetura da *RNN* consiste numa célula recorrente simples com *activation Tanh*, que processa os *embeddings* palavra a palavra. A previsão é baseada exclusivamente na última saída da sequência, à qual se aplica uma camada densa com *activation Sigmoid*, produzindo uma probabilidade entre 0 e 1.

Durante o treino, utilizámos o algoritmo de *Backpropagation Through Time (BPTT)* com *truncation* e otimizámos os pesos utilizando *Stochastic Gradient Descent (SGD)* com *momentum*, ajustando-os com base na *loss function Binary Cross-Entropy*. O modelo foi treinado com diferentes combinações de *hiperparâmetros* (epochs, batch size, learning rate, etc.) até obter a melhor configuração.

Apesar de não termos utilizado bibliotecas como *TensorFlow* ou *PyTorch*, a nossa implementação demonstrou capacidade de generalização, conseguindo capturar dependências temporais relevantes nos textos. A avaliação do modelo no conjunto de teste revelou um bom desempenho, com os seguintes resultados:

Tabela 3: Avaliação das métricas no *dataset* de teste *merged_inputs* - RNN

Metric	**Value**
Accuracy	80.85%
Precision	71.27%
Recall	98.47%
F1-Score	82.69%
Balanced Accuracy	82.01%

3.2 Modelos de Deep Learning e implementação baseada em Tensorflow

3.2.1 Gated Recurrent Units (GRU)

O modelo baseado em Gated Recurrent Units (*GRU*) foi implementado com o objetivo de classificar textos como sendo gerados por inteligência artificial ou escritos por humanos. O GRU é uma variante das Recurrent Neural Networks (*RNN*) que se destaca pela sua capacidade de capturar dependências temporais em sequências de dados, tornando-o particularmente adequado para tarefas de processamento de linguagem natural.

A arquitetura do modelo incluiu uma camada de *embedding* para transformar palavras em representações vetoriais densas, seguida de uma camada

GRU com 64 unidades e uma única camada recorrente, que foi responsável por extrair padrões temporais nos textos. Posteriormente, uma camada densa com ativação *softmax* foi utilizada para classificar os textos em duas categorias: "IA"ou "Humano". Para evitar *overfitting*, foi empregada a regularização por meio da técnica de *dropout* com uma taxa de 0.1.

O pré-processamento dos dados envolveu a remoção de caracteres especiais e de *stopwords*, além da lematização para reduzir a dimensionalidade do vocabulário. Os textos foram *tokenized* e convertidos em sequências numéricas, utilizando um vocabulário limitado a 1000 palavras. Durante o treino, utilizou-se a função de perda *cross-entropy* e o otimizador *Adam*, com uma taxa de aprendizagem de 0.001. Além disso, a técnica de *Early Stopping* foi aplicada para interromper o processo caso não houvesse melhorias significativas na validação.

Os melhores *hyperparameters* encontrados foram uma dimensão de *embedding* de 32, 64 unidades na camada GRU, uma única camada recorrente, 8 épocas de treino, um *batch size* de 8, uma taxa de aprendizagem de 0.001 e uma taxa de *dropout* de 0.2. Durante o treino, o modelo atingiu uma *accuracy* de 99% no conjunto de treino, com uma perda de 0.2331, enquanto a melhor *accuracy* no conjunto de validação foi de 96.21%.

Toda esta informação, incluindo a evolução da curva de aprendizagem do modelo, está representada graficamente na Figura 1.

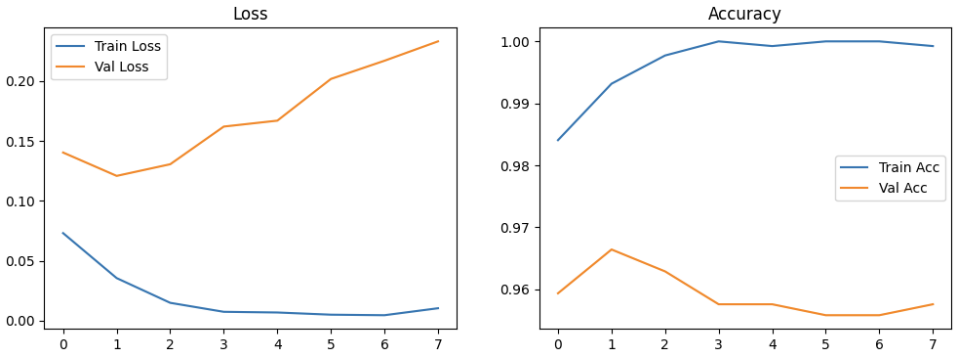

Figura 1: Curva de aprendizagem do modelo GRU.

Tabela 4: Avaliação das métricas no *dataset* de teste *merged_inputs* - GRU

Metric	**Value**
Accuracy	95.76%
Precision	94.50%
Recall	97.17%
F1-Score	95.82%

3.2.2 Robustly Optimized BERT Pretraining Approach (RoBERTa)

Uma outra abordagem implementada pelo grupo consiste num modelo de classificação utilizando o *Transformer* RoBERTa.

O grupo integrou os conjuntos de treino e teste, efetuou a limpeza dos textos e optou por manter a pontuação e a numeração, pois estes preservam informações essenciais, como pausas, entoações e hierarquias.

Tendo os dados tratados, é então inicializado o *tokenizer* do modelo RoBERTa e é efetuada a tokenização dos textos. A tokenização é realizada sem truncamento ou *padding* para determinar o comprimento dos *tokens*, sendo posteriormente calculado o comprimento máximo baseado no percentil de 90 dos comprimentos das sequências. Este comprimento é então utilizado para tokenizar novamente os textos, aplicando truncamento e *padding*, de forma a garantir que todas as entradas têm o mesmo tamanho. Os dados tokenizados são convertidos num novo *Dataset* e posteriormente divididos em *batches*, de forma a facilitar o treino do modelo.

O modelo é carregado a partir do checkpoint *"roberta-base"* e são ajustados os parâmetros de dropout para a *attention layer* e para a camada de embeddings, contribuindo para aumentar a regularização e ajudar a prevenir *overfitting*. O otimizador Adam é configurado com uma *learning rate* que decai exponencialmente, e a função de perda utilizada é a *SparseCategoricalCrossentropy*, a qual trabalha com os *logits* (os outputs brutos do modelo). Durante o treino, são ainda utilizados *callbacks*, como o *EarlyStopping*, para monitorizar a *loss* e restaurar os melhores pesos.

Foi ainda explorado o uso da GPU como acelerador, que se revelou fundamental para lidar com a natureza intensiva em computação dos modelos *Transformer*, reduzindo significativamente o tempo de treino e facilitando a realização de mais experiências de *fine-tuning* num intervalo de tempo razoável. Para lidar com a aleatoriedade ao longo do processo, definimos a mesma *seed* em vários pontos do código, incluindo o módulo `random`, a biblioteca NumPy e o próprio TensorFlow.

Com 3 *epochs* (de 5 epochs definidas inicialmente) de treino, o modelo RoBERTa atingiu uma **accuracy** de 99.29% e uma **loss** de 0.0272.

Tabela 5: Avaliação das métricas no *dataset* de teste *merged_inputs* - RoBERTa

Metric	**Value**
Accuracy	99.29%
Precision	98.95%
Recall	99.65%
F1-Score	99.30%
Balanced Accuracy	99.30%

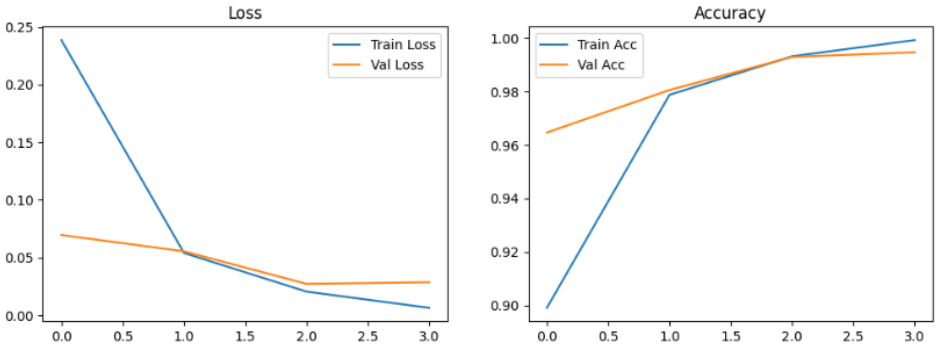

Figura 2: Curva de aprendizagem do Transformer RoBERTa

3.2.3 Bidirectional Encoder Representations from Transformers (BERT)

O *BERT* (Bidirectional Encoder Representations from Transformers) é um modelo de linguagem baseado na arquitetura *Transformer*, desenvolvido pela Google, que revolucionou o processamento de linguagem natural (*NLP*) ao permitir a análise contextualizada bidirecional de textos. Para tarefas de classificação de sentenças, o *BERT* é *fine-tuned* num conjunto de dados rotulados, utilizando a representação contextualizada da sentença (especialmente o token `[CLS]`) para inferir a classe desejada. A biblioteca *Hugging Face* facilita a implementação do *BERT* através da sua API *Transformers*, oferecendo modelos pré-treinados como o `bert-base-uncased` e destacando o processo de ajuste fino e avaliação. Outro motivo para a sua escolha foi que, devido a dificuldades em obter dados em larga escala e de qualidade, a característica de ser um modelo pré-treinado, necessitando apenas de *fine-tuning*, revelou-se bastante útil.

Por ser uma abordagem pré-treinada, há pouca margem para otimização de *hyperparameters*; no entanto, alguns dos utilizados foram um *batch size* de 16 e uma *learning rate* de 1e-5, com alguns *callbacks* para reduzi-la conforme as *epochs* e/ou a *validation accuracy*, bem como um *EarlyStop* para o treino.

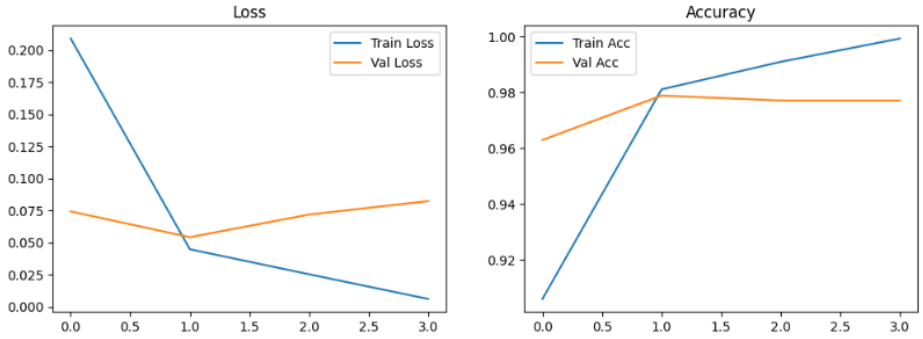

Figura 3: Curvas de aprendizagem do modelo BERT (`bert-base-uncased`)

As curvas indicam um certo *overfit* mas ainda sim os resultados de validação foram relativamente satisfatórios, provavelmente um número maior de instâncias de treino levaria a uma melhor generalização do modelo.

Tabela 6: Avaliação das métricas no *dataset* de teste *merged_ inputs* - BERT

Metric	**Value**
Accuracy	97.88%
Precision	99.28%
Recall	97.17%
F1-Score	98.21%
Balanced Accuracy	98.23%

3.2.4 Long Short-Term Memory (LSTM)

O modelo baseado em LSTM (Long Short-Term Memory) foi desenvolvido com o intuito de classificar textos como sendo gerados por inteligência artificial ou escritos por humanos. A arquitetura foi concebida para capturar dependências de longo prazo nos dados, característica essencial no processamento de linguagem natural.

A pipeline de pré-processamento envolveu a limpeza do texto (remoção de pontuação, normalização e *stopwords*) e posterior tokenização. Em vez de utilizar tokens simples, o modelo recorreu a *embeddings* treináveis (com 100 dimensões), sendo aplicadas sequências com *padding* fixo de 120 tokens.

O treino do modelo utilizou o otimizador *Adam* com uma taxa de aprendizagem de 0.001 e a função de perda *Binary Crossentropy*. O conjunto de dados foi dividido em 70% para treino e 30% para teste. Foram exploradas várias combinações de *hyperparameters*, tendo-se identificado a seguinte como a melhor configuração: 4 *epochs*, *batch size* de 16, *learning rate* de 0.001, *LSTM units* = [32, 32] e *dropout* = 0.1.

Tabela 7: Avaliação das métricas no *dataset* de teste *merged_ inputs* - LSTM

Metric	**Value**
Accuracy	98.37%
Precision	98.35%
Recall	98.35%
F1-Score	98.35%
Balanced Accuracy	98.35%

4 Resultados obtidos

Tabela 8: Tabela de resultados dos modelos na previsão do dataset `submission3_inputs.csv`

Modelo	**Accuracy (%)**
LR	56
DNN	63
RNN	60
GRU	66
RoBERTa	89
BERT	75
LSTM	67

Como se pode observar na Tabela 8, a escolha dos modelos a submeter foi acertada, uma vez que foram selecionados os dois modelos com melhor desempenho em termos de **accuracy** (RoBERTa e BERT). Além disso, entre a submissão inicial e a versão final, verificou-se uma melhoria nos resultados obtidos. O modelo BERT, que nos resultados da submissão 3 apresentava uma **accuracy** de 73%, passou para 75% após melhoria de código, enquanto que o modelo RoBERTa melhorou a **accuracy** de 86% para 89% após passar pela mesma melhoria. Estes resultados colocariam o nosso grupo na terceira posição da classificação final.

5 Conclusão

5.0.1 Resultados e Perspetivas

Os resultados obtidos demonstram um desempenho positivo das abordagens seguidas, com uma taxa de acerto de 67% nos modelos desenvolvidos de raiz e de 86% nos modelos baseados em *TensorFlow*, o que evidencia a eficácia

das soluções implementadas e a solidez dos frameworks utilizados para *training* e *inference*. Em comparação com os restantes grupos, alcançámos o 3.º lugar com os nossos modelos próprios (*RNN* e *DNN*) e o 4.º lugar na primeira e segunda submissão com modelos baseados em *TensorFlow* (*BERT* e *RoBERTa*).

Durante o desenvolvimento, enfrentámos desafios como o ajustamento de *hyperparameters*, o equilíbrio entre precisão e generalização, e a limitação dos dados disponíveis. Algumas melhorias possíveis passariam por uma afinação mais cuidadosa dos hiperparâmetros para reduzir o *overfitting*, e pelo aumento do conjunto de dados para treinos mais representativos. A nível de arquitetura, poderíamos ter explorado soluções híbridas ou mesmo abordagens mais recentes como *few-shot* ou *zero-shot learning*, usando modelos como *Claude*, *GPT-3* ou *FLAN-T5*, que já demonstraram bons resultados em tarefas de classificação com pouca supervisão.