Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Correção automática de redações do ENEM usando aprendizado de máquina

Mateus Latrova Stephanin

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Dênis Deratani Mauá
Cossupervisor: Igor Cataneo Silveira

São Paulo
2023

O que ama a correção ama o conhecimento, mas o que aborrece a repreensão é um bruto (Provérbios 12:1)

Agradecimentos

Exaltar-te-ei, ó Senhor, porque tu me exaltaste; e não fizeste com que meus inimigos se alegrassem sobre mim.
— Salmos 30:1

Primeiramente, agradeço a Deus por sempre viver junto a Ele, e por ter me dado força, saúde, inteligência e capacidade para chegar até este momento. Sem ele, jamais teria capacidade de entrar neste curso, nem de superar os obstáculos que aqui enfrentei, nem de chegar ao final dele. Graças te dou, ó Senhor, pois és digno de toda a honra, todo o louvor e toda a gratidão.

Agradeço também à minha amada esposa Claudia Jemima, a qual desde o começo foi uma companheira que me apoiou, e nos momentos mais difíceis, nos quais pensei em desistir, me falou palavras que me deram força para continuar.

Também, agradeço aos meus pais, Evandro e Andrea, pelo imensurável amor que me proporcionaram, pelo ensino de princípios e valores maravilhosos que me acompanham desde a minha infância, por terem lutado para que eu alcançasse aquilo que eles não tiveram a oportunidade de alcançar, e por serem a minha fortaleza em todos os momentos da vida.

Agradeço ao meu avô, sem o qual não seria possível chegar até aqui, por todo o suporte que deu a mim e à minha família, por todos os conselhos e ensinamentos, os quais guardarei para sempre no meu coração.

Agradeço ao meu orientador Igor Cataneo, por ter me ajudado e guiado com muita paciência e didática.

Por fim, agradeço aos colegas e professores do IME por todos os momentos e por todo o aprendizado que aqui obtive, os quais levarei para a minha vida.

Resumo

Mateus Latrova Stephanin. **Correção automática de redações do ENEM usando aprendizado de máquina**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Avaliadores automáticos de redações são softwares que anotam textos escritos em linguagem natural com pontuações que refletem objetivos educacionais. Eles portanto fornecem uma maneira barata e eficiente de assistir na aprendizagem, especialmente para o caso de redações em exames padronizados, como o caso do ENEM ou do TOEFL. Tal tarefa pode ser enxergada pela lente da classificação automática de textos e abordada por técnicas de aprendizado de máquina. Este projeto de conclusão de curso visa a investigação de modelos de linguagem baseados em redes neurais profundas para a avaliação automática de redações do ENEM. Mais especificamente, o foco da avaliação foi a segunda competência cobrada na redação do ENEM: "Compreender o tema e não fugir do que é proposto". Foram implementados dois modelos para avaliarem essa competência: um para detectar se a redação foge ao tema proposto, e outro para detectar se a redação é ou não um aglomerado de palavras.

Palavras-chave: Classificação automática de textos. Transformers. Aprendizado profundo. ENEM. Redação. Fuga ao tema. Aglomerado de palavras.

Abstract

Mateus Latrova Stephanin. **Automatic correction of essays from ENEM using machine learning**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Automatic essay evaluators are software that annotates texts written in natural language with a score that reflects educational goals. Thus, they bring an efficient and cheap way of assisting in learning, especially in the case of essays in standardized exams, as in the case of ENEM or TOEFL. Such a task can be seen through the lens of automatic text classification and approached by machine learning techniques. This capstone project aims to investigate language models based on deep neural networks for the automatic correction of ENEM essays. More specifically, the correction focus was the second competence demanded in ENEM essays: "Understand the topic and do not deviate from what is proposed". Two models were implemented to evaluate that competence: one to detect if an essay deviates from the proposed topic, and another one to detect if it is or not a cluster of words.

Keywords: Automatic text classification. Transformers. Deep learning. ENEM. Essay. Topic deviation. Cluster of words.

Lista de abreviaturas

IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo
ENEM	Exame Nacional do Ensino Médio
INEP	Instituto Nacional de Estudos e Pesquisas Educaionais Anísio Teixeira
IA	Inteligência Artificial
PLN	Processamento de Linguagem Natural
BNCC	Base Nacional Comum Curricular
GPU	Unidade de processamento gráfico (*Graphic Processing Unit*)
TPU	Unidade de processamento de tensores (*Tensor Processing Unit*)

Lista de figuras

1.1 Número de inscritos no ENEM ao longo dos últimos anos. Fonte: G1 . . . 2

2.1 Grade específica da competência II. Fonte: INEP . . . 6
2.2 Arquitetura de um modelo transformer (Vaswani *et al.*, 2017) . . . 8
2.3 Exemplo de embeddings calculadas num corpus voltado a análise de sentimento . . . 8
2.4 Exemplo de matriz de confusão . . . 11

4.1 Perplexidade por redação . . . 39
4.2 Média da perplexidade das redações por grupo de nota . . . 40
4.3 Desvio-padrão da perplexidade das redações por grupo de nota . . . 40
4.4 Matrizes de confusão - *dataset A* - teste 1 . . . 42
4.5 Matriz de confusão - confiança 75% - *dataset B* - teste 1 . . . 43
4.6 Matriz de confusão - confiança 90% - *dataset B* - teste 1 . . . 43
4.7 Matriz de confusão - confiança 90% - *dataset B* - teste 2 . . . 44

Lista de tabelas

4.1 Acurácia e F_1 nos *datasets* de validação e teste . . . 33
4.2 Intervalos de confiança para classificação de aglomerados - teste 1 . . . 41
4.3 Intervalos de confiança para classificação de aglomerados - teste 2 . . . 44

Lista de programas

4.1 Dicionário para renomear colunas do *dataset* de redações. 18
4.2 Método para pré-processar o texto da redação 18
4.3 Método para criar coluna do rótulo de fuga ao tema 19
4.4 Métodos para criar dados artificiais 20
4.5 Colunas a serem removidas do *dataset* de redações 21
4.6 Método para a criação do *dataset* de teste 22
4.7 Método para concatenar textos do tema 23
4.8 Classe que combina os *datasets* de redações e de temas 24
4.9 Classe da rede neural treinada para detecção de fuga ao tema 26
4.10 Método construtor do *fine-tuner* 28
4.11 Método de execução do treinamento do modelo de detecção de fuga ao tema 29
4.12 Método construtor do preprocessador de redações 35
4.13 Método _preprocess_dataset 35
4.14 Mapeamento para renomeação das colunas do *dataset* de redações . . . 36
4.15 Método que cria coluna com a nota da segunda competência 36
4.16 Atributos estáticos do calculador de perplexidade 37
4.17 Método construtor do calculador de perplexidade 37
4.18 Método que calcula a perplexidade de todas as redações de um *dataset* . 38
4.19 Método que calcula a perplexidade de uma redação 39

Sumário

1 Introdução **1**
1.1 Contextualização e motivação . . . 1
1.2 Objetivos . . . 3
1.3 Organização do texto . . . 4

2 Fundamentação teórica **5**
2.1 Correção de redações do ENEM . . . 5
2.1.1 A competência II . . . 5
2.2 Modelos *transformers* . . . 7
2.2.1 BERT e BERTimbau . . . 9
2.2.2 *Fine-tuning* . . . 9
2.2.3 Perplexidade . . . 9
2.2.4 Métricas de avaliação de modelos . . . 10
2.3 Ferramentas de programação . . . 11
2.3.1 Python . . . 11
2.3.2 *Jupyter Notebooks* . . . 12
2.3.3 *Hugging Face* . . . 13

3 Datasets **15**
3.1 Datasets de redações . . . 15
3.2 Dataset de temas . . . 16

4 Desenvolvimento **17**
4.1 Detecção de fuga ao tema . . . 17
4.1.1 Hipótese . . . 17
4.1.2 Pré-processamento dos dados . . . 17
4.1.3 Treinamento do modelo . . . 25
4.1.4 Resultados da avaliação e teste do modelo . . . 32
4.2 Modelo de detecção de aglomerado de palavras . . . 34

4.2.1 Hipótese 34
4.2.2 Pré-processamento dos dados 34
4.2.3 Calculador de perplexidade 36
4.2.4 Validação da hipótese 39
4.2.5 Estratégia de detecção e seus resultados 41

5 Considerações finais **45**

Referências **47**

Capítulo 1

Introdução

1.1 Contextualização e motivação

No Brasil, os jovens estudantes possuem algumas opções como porta de entrada para a universidade. Porém, a maior e mais abrangente delas é, de longe, o Exame Nacional do Ensino Médio(ENEM), o qual acontece anualmente na maioria das escolas em todos os estados do país.

Nesse vestibular, assim como na maioria dos outros, além de serem cobrados conteúdos presentes na Base Nacional Comum Curricular(BNCC) como aqueles voltados às áreas de Linguagens, Ciências Humanas, Naturais e Matemática, também é esperado que o candidato redija um texto dissertativo-argumentativo posicionando-se a respeito de algum tema o qual é revelado no momento em que o exame se inicia.

A natureza desse texto, por si só, revela a sua intenção: fazer com que o candidato mostre os conhecimentos obtidos ao longo de seus estudos, além de articular e relacioná-los de forma analítica e crítica a fim de defender o seu ponto de vista.

Entretanto, por ter um limite de tamanho pequeno(trinta linhas) e também por ter critérios de correção muito bem definidos e explicados, os estudantes são orientados a seguir uma estrutura padrão no momento da redação, composta por três partes com finalidades distintas: introdução, desenvolvimento e conclusão.

Na introdução, o estudante deve, principalmente, contextualizar o leitor com o tema a ser tratado de uma forma criativa e interessante. Porém, outro ponto muito importante é deixar claro qual é a sua tese, ou seja, o seu posicionamento em relação ao tema em questão.

Em seguida, na parte do desenvolvimento, ele deve apresentar uma sequência de argumentos lógicos que justifiquem e defendem a sua tese, com base em qualquer conhecimento de referência. Por fim, no momento da conclusão, é esperado que o aluno retome a sua tese, e proponha uma possível melhoria ou solução para os problemas que fazem parte daquele tema, de forma que ela seja coerente com o seu posicionamento.

Tendo em vista essa padronização da escrita, a correção desses textos busca avaliá-los em cinco principais aspectos (Educação, 2012):

1) Domínio da norma padrão da língua escrita;

2) Compreensão da proposta de redação e aplicação de conceitos das várias áreas do conhecimento para o desenvolvimento do tema nos limites estruturais do texto dissertativo-argumentativo;

3) Capacidade de selecionar, relacionar, organizar e interpretar informações, fatos, opiniões e argumentos em defesa de um ponto de vista;

4) Conhecimento dos mecanismos linguísticos necessários à construção da argumentação;

5) Elaboração de proposta de intervenção para o problema abordado, respeitados os direitos humanos.

Mesmo com uma estrutura de correção muito bem pontuada, é perfeitamente possível que, dados dois corretores diferentes, a nota para cada critério e a nota final da redação dadas por ambos sejam diferentes. Esse tipo de situação ocorre devido à natureza subjetiva do ser humano.

Além disso, também há um aspecto importante que é a imensa quantidade de provas e redações a serem corrigidas. Como se pode observar na figura 1, é comum haver alguns milhões de inscritos no ENEM, e, portanto, de redações a serem corrigidas também. Por conta disso, é necessário que haja um grande custo a fim de contratar cerca de 3 mil professores que realizem a correção ao longo de aproximadamente um mês(Albuquerque, 2023).

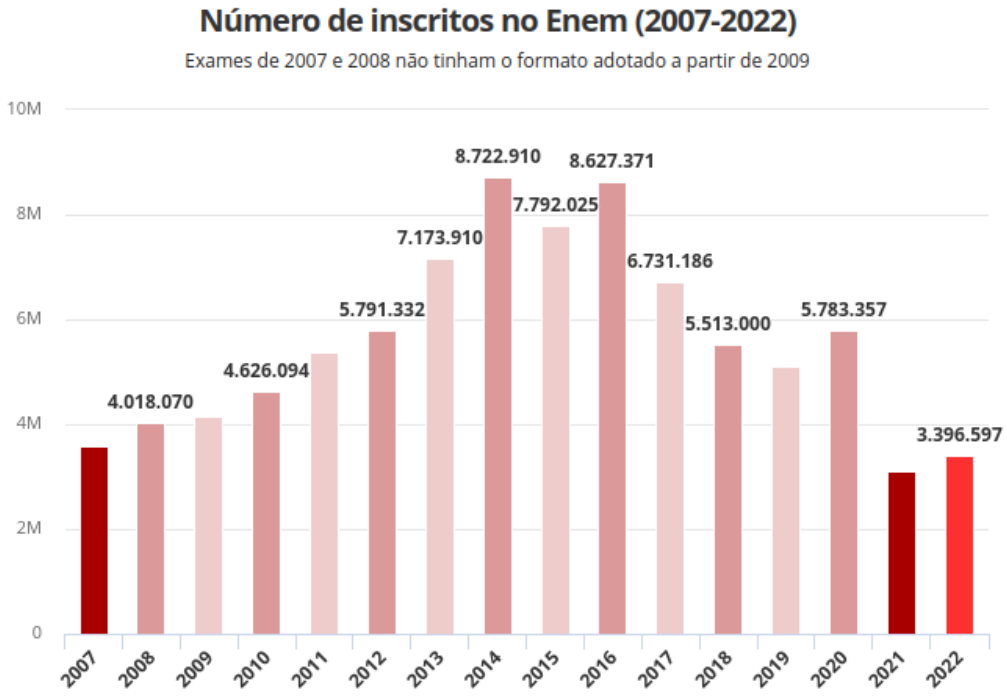


Figura 1.1: *Número de inscritos no ENEM ao longo dos últimos anos. Fonte: G1*

Dessa forma, a natureza objetiva e imparcial da ciência e da tecnologia que existem hoje pode ajudar a alcançarmos uma correção automática, a qual será mais uniforme e coerente.

Embora seja um grande desafio técnico, as consequências disso podem ser muito benéficas em relação ao custo e à eficiência na correção das redações, pois poderia-se economizar o dinheiro pago a esses professores, e também transformar a correção em uma tarefa mais ágil e escalável devido à rapidez proporcionada por uma correção automática e também à possibilidade de executar diversas instâncias de corretores automáticos simultaneamente.

1.2 Objetivos

Assim, devido à característica intrínseca de reconhecer padrões presente em modelos de inteligência artificial que utilizam técnicas de aprendizado de máquina, o foco deste trabalho foi a experimentação desses modelos na tarefa de corrigir redações do ENEM. Ou seja, dada uma redação, atribuir-lhe uma nota.

Porém, a correção de uma redação do ENEM não é uma tarefa simples. Como pode se observar na seção anterior, há cinco competências nas quais ela deve ser avaliada. Devido à complexidade presente em cada uma dessas competências, espera-se que um único modelo que tente captar a nota final de uma redação utilizando-se de uma abordagem genérica não tenha um desempenho ótimo.

Por conta disso, depois de analisar todas as competências, foi decidido que a competência com a qual trabalharíamos seria a de número dois: "Compreensão da proposta de redação e aplicação de conceitos das várias áreas do conhecimento para o desenvolvimento do tema nos limites estruturais do texto dissertativo-argumentativo". Essa competência foi escolhida, pois foram identificados nela alguns aspectos que abrem a possibilidade para que técnicas de aprendizado de máquina tragam um bom desempenho.

Nesse sentido, este trabalho de conclusão de curso buscou, essencialmente, a compreensão dos principais fundamentos e algoritmos que compõem o NLP (subárea da IA) mediante uma preparação técnica sobre esse assunto, e a implementação e curadoria de modelos de redes neurais profundas que fossem capazes de corrigir redações sob o prisma da competência dois.

A fim de obter esses conhecimentos teóricos, foi lida a primeira parte do livro "Speech and Language Processing"(terceira edição), escrito por Dan Jurafsky e James H. Martin, a qual é composta por doze capítulos(**jurafsky**).

Além disso, buscando aprimorar e aplicar os conhecimentos obtidos pelo livro, foi seguido o tutorial da biblioteca open-source *Hugging Face* (**hugging**). Essa biblioteca escrita na linguagem Python fornece diversos recursos para desenvolvedores se familiarizarem com códigos que resolvem problemas voltados ao processamento de linguagem natural, como, por exemplo, a classificação de textos.

Em seguida, foram implementadas duas estratégias de correção utilizando os conceitos e técnicas aprendidos anteriormente, como, por exemplo, a perplexidade (2.2.3) e o *fine-tuning*(2.2.2) de modelos transformers pré-treinados.

Por fim, foram feitos diversos experimentos envolvendo a parametrização dessas estratégias, buscando aumentar a sua performance, ou seja, melhorar a qualidade da correção automática das redações.

1.3 Organização do texto

O texto deste trabalho será organizado da seguinte forma: primeiro, será feita uma explicação sobre os conceitos e ferramentas fundamentais para o entendimento dos experimentos e seus resultados. Os conceitos explorados envolvem a área da Computação, Matemática, Língua Portuguesa e também especificidades do contexto da correção do ENEM.

Após essa fundamentação, será descrito como se deu o desenvolvimento do trabalho. Serão explicados os detalhes da lógica da implementação e também dos resultados alcançados em cada um dos experimentos. Por fim, haverá uma interpretação a respeito dos resultados obtidos e a indicação de possíveis próximos passos para seguirem este trabalho.

Capítulo 2

Fundamentação teórica

Para se obter um entendimento mais profundo dos experimentos realizados ao longo deste projeto, faz-se necessário construir uma fundamentação teórica primeiro. Neste capítulo, buscou-se explicar com clareza todos os conceitos essenciais à compreensão da implementação, incluindo conceitos da área da Computação(desenvolvimento de software e Inteligência Artificial) e também conceitos relacionados à Língua Portuguesa, a fim de que se entenda também qual será a aplicação prática deste projeto.

2.1 Correção de redações do ENEM

A redação do ENEM é um exame altamente padronizado, e a sua correção manual envolve a análise criteriosa de cinco competências, cada uma com diversas especificidades. Como foi introduzido na seção 1.1, o foco da correção dentro deste projeto foi na competência dois: "Compreensão da proposta de redação e aplicação de conceitos das várias áreas do conhecimento para o desenvolvimento do tema nos limites estruturais do texto dissertativo-argumentativo"(Educação, 2012). Nesta seção, serão explicados os conceitos fundamentais que um corretor precisa conhecer a fim de ser capaz de corrigir uma redação dentro dessa competência, levando sempre em consideração os aspectos que são mais relevantes para ela.

2.1.1 A competência II

Todas as competências da redação do ENEM podem ser avaliadas em seis níveis: 0, 1, 2, 3, 4 e 5. Cada um desses níveis reflete uma nota obtida pelo candidato, as quais são, respectivamente: 0, 40, 80, 120, 160 e 200. Além disso, em toda a redação do ENEM, há uma proposta de redação, a qual dita o tema que deverá ser abordado pelo candidato, assim como levanta questionamentos que o candidato deverá responder ao longo da sua redação defendendo o seu próprio ponto de vista.

Nessa competência, os pontos principais analisados pelo corretor são o tema e a tipologia textual. Dessa forma, para cada nível de avaliação, leva-se em conta a qualidade da redação em relação a esses dois aspectos simultaneamente. Ela é muito importante, pois

é a partir desses pontos que as outras competências se orientam e organizam (INEP, 2020).

Para se entender o que exatamente leva uma redação a ser avaliada em um determinado nível dentro dessa competência levando em consideração os aspectos mencionados, será apresentada na figura 2.1 a seguir a grade específica da competência II. De forma geral, o que se busca avaliar em relação ao tema, é se o candidato compreendeu o tema que deve ser abordado e também o contexto da proposta em toda a sua amplitude.

COMPETÊNCIA II Compreender a proposta de redação e aplicar conceitos das áreas de conhecimento, dentro dos limites do texto dissertativo-argumentativo em prosa					
1	Tangência ao tema	**OU**	• Texto composto por aglomerado de palavras **OU** • Traços constantes de outros tipos textuais		
2	Abordagem completa do tema	**E**	• 3 partes do texto (2 delas embrionárias) **OU** • Conclusão finalizada por frase incompleta	Textos que apresentam muitos trechos de cópias dos textos motivadores não devem ultrapassar esse nível	
3	Abordagem completa do tema	**E**	3 partes do texto (1 parte pode ser embrionária)	**E**	• Repertório baseado nos textos motivadores **E/OU** • Repertório não legitimado **E/OU** • Repertório legitimado, **MAS** não pertinente ao tema
4	Abordagem completa do tema	**E**	3 partes do texto (nenhuma delas embrionária)	**E**	Repertório legitimado **E** pertinente ao tema, **MAS** com uso improdutivo
5	Abordagem completa do tema	**E**	3 partes do texto (nenhuma delas embrionária)	**E**	Repertório legitimado E pertinente ao tema, **COM** uso produtivo

Figura 2.1: *Grade específica da competência II. Fonte: INEP*

Níveis de avaliação

Como se pode observar nessa figura, para cada nível de avaliação, há um conjunto de características que se espera que a redação tenha, sendo que na coluna da esquerda estão as características voltadas à abordagem do tema, e na coluna da direita, as voltadas para o

uso e a estrutura do tipo textual dissertativo-argumentativo.

Neste trabalho, foram abordados apenas os níveis 0 e 1. O nível 0, o qual não está presente na figura acima, é o primeiro cujos critérios são avaliados. Para que uma redação obtenha o nível 0, é necessário que ela tenha fugido completamente do tema proposto, isto é, que ela tenha falado de um assunto que não está presente na proposta e nos textos motivadores.

Em seguida, caso a redação não tenha fugido do tema, avalia-se se a redação pertence ao nível 1. Como se observa na figura 2.1, a redação é avaliada no nível 1 quando ela tange o tema, ou quando o texto é composto por um aglomerado de palavras, ou possui traços constantes de outros tipos textuais. Dentre esses três critérios, neste trabalho abordou-se apenas o segundo. Um aglomerado de palavras consiste em um conjunto de palavras que são justapostas, porém não necessariamente constroem um significado compreensível. Qualquer pessoa que lê um texto composto por um aglomerado de palavras provavelmente terá dificuldade de entender a ideia que ali foi expressa.

2.2 Modelos *transformers*

Agora, o foco sairá da Língua Portuguesa e irá para a área da Inteligência Artificial(IA), mais especificamente, dentro da subárea do aprendizado de máquina no contexto do Processamento de Linguagem Natural(PLN). Os modelos *transformers* pré-treinados têm alterado drasticamente o cenário do PLN e diversas outras aplicações em IA, trazendo uma grande aceleração de desenvolvimento de novos modelos. Por conta disso, aqui será explicado um pouco da história, do funcionamento e do impacto dessa tecnologia no âmbito do PLN, que é o foco deste trabalho.

Os modelos *transformers* pré-treinados têm se estabelecido como um dos pilares do PLN moderno, exibindo desempenho excepcional em diversas tarefas. Eles trazem uma nova e sofisticada arquitetura de rede neural, apresentada na figura 2.2, e surgiram como resultado do trabalho reportado no artigo *"Attention is All You Need"* (Vaswani *et al.*, 2017). Este artigo inovador propôs um mecanismo de autoatenção, base para os *transformers*, permitindo paralelização eficiente de recursos e a captura do conhecimento de um contexto específico.

Tais modelos, assim como os demais, não recebem diretamente palavras como entrada, mas sim, representações vetoriais destas, chamadas *embeddings*. Essas representações são utilizadas em toda aplicação de PLN que precise se aproveitar do significado das palavras (Jurafsky e Martin, 2020).

Um embedding é um ponto num espaço multidimensional, calculado com base na distribuição de palavras em um corpo textual (Jurafsky e Martin, 2020). A hipótese para se fazer isso desta forma é que acredita-se que há uma relação entre a distribuição das palavras num determinado contexto e a sua similaridade.

Uma ilustração da validação experimental dessa hipótese está na figura 2.3 abaixo, retirada de Jurafsky e Martin, 2020, na qual os *embeddings* de palavras que trazem um mesmo sentimento estão próximas umas das outras num espaço bidimensional.

Uma das grandes inovações trazidas na arquitetura dos *transformers* foi a capacidade

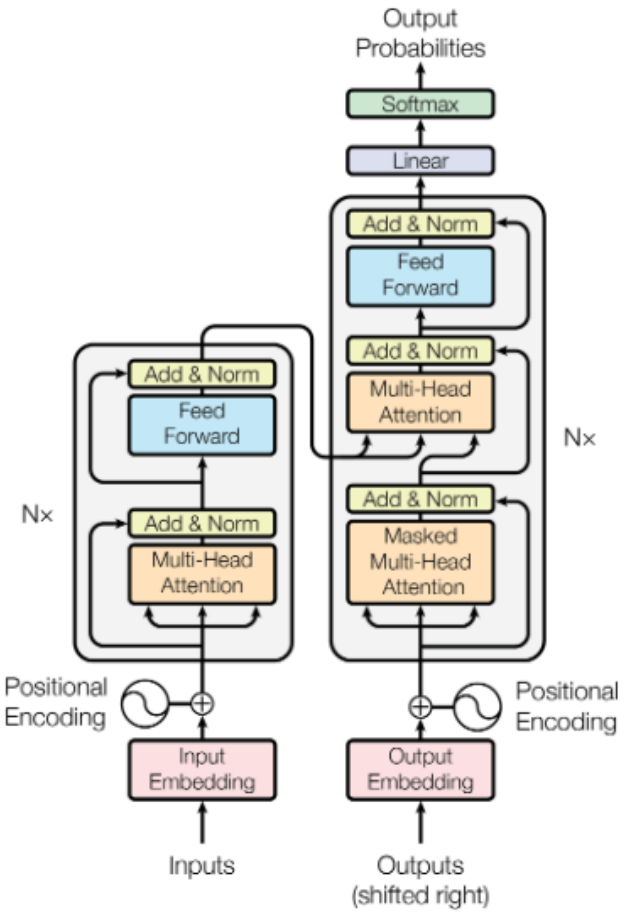


Figura 2.2: *Arquitetura de um modelo transformer (Vaswani et al., 2017)*

Figura 2.3: *Exemplo de embeddings calculadas num corpus voltado a análise de sentimento*

de capturar dependências contextuais distantes em frases por meio de mecanismos de autoatenção. O modelo *transformer* consiste em camadas de codificação e decodificação, as quais fizeram com que ele superasse as redes neurais recorrentes (Lipton *et al.*, 2015) tradicionais na manipulação de dados sequenciais.

2.2.1 BERT e BERTimbau

Em 2018, ano seguinte ao surgimento dos modelos *transformers*, foi criado o modelo *transformer* BERT (Devlin *et al.*, 2019), sigla para *Bidirectional Encoder Representations from Transformers*, o qual marcou um avanço significativo ao introduzir o pré-treinamento bidirecional, possibilitando que os modelos considerem o contexto à esquerda e à direita da palavra sobre a qual está o foco atual durante o treinamento.

Os dois modelos pré-treinados usados neste trabalho são baseados nesse modelo, mais especificamente, na sua versão em português, o BERTimbau (Souza *et al.*, 2020).

2.2.2 *Fine-tuning*

O processo de *fine-tuning*, cuja eficácia foi mostrada no artigo em que foi apresentado o modelo BERT (Devlin *et al.*, 2019), é uma técnica essencial no campo do aprendizado de máquina, particularmente quando se trabalha com transferência de aprendizagem (Ruder *et al.*, 2019). Essa abordagem envolve treinar um modelo que já foi pré-treinado em uma grande quantidade de dados para uma tarefa específica ou domínio, utilizando um conjunto de dados menor e mais específico. O *fine-tuning* permite que modelos pré-treinados se adaptem de uma forma muito eficiente a novos contextos, tarefas ou domínios, aproveitando o conhecimento prévio adquirido durante o treinamento inicial.

Para aplicá-lo, é comum usar a arquitetura de um modelo pré-treinado como ponto de partida e ajustar seus pesos durante o treinamento com dados específicos da tarefa desejada, por exemplo, dados de redações a fim de detectar a fuga ao tema da redação (caso do primeiro experimento deste trabalho). Geralmente, o conjunto de dados de treinamento para o *fine-tuning* é menor do que o conjunto de dados original usado no pré-treinamento. Isso é especialmente útil quando há limitações de recursos computacionais, pois, assim, elas não impedem o processamento desses dados, ou quando o conjunto de dados específico da tarefa é pequeno, não sendo possível treinar um modelo do zero apenas com esses poucos dados.

2.2.3 Perplexidade

A perplexidade é uma métrica muito utilizada para avaliar modelos de linguagem. Dado um conjunto de palavras $W = w_1 w_2 ... w_N$, a perplexidade de um modelo no conjunto W é o inverso da probabilidade do conjunto normalizada pelo tamanho do conjunto, e é dada pela fórmula (Jurafsky e Martin, 2020):

$$\text{PPL(W)} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_1 w_2 ... w_N)}}$$

Assim, quanto maior a probabilidade condicional do conjunto W, menor será o valor dessa métrica. E quanto menor essa probabilidade, maior a perplexidade.

2.2.4 Métricas de avaliação de modelos

No contexto de modelos de inferência em PLN, o uso de algumas métricas é muito comum a fim de compreender melhor o desempenho de um modelo de classificação de textos em todas as possíveis situações nas quais ele pode ser colocado. Pelo fato de este trabalho lidar apenas com a classificação binária, ou seja, aquela em que o modelo pode atribuir a um dado de *input* apenas uma categoria entre duas possíveis (comumente, chamadas de positiva e negativa), tratar-se-á apenas de métricas voltadas a essa situação.

Dentre elas, aqui serão apresentadas a acurácia, a precisão, a *recall* e a F_1. Além disso, será feita uma explicação a respeito de uma ferramenta muito utilizada com esse mesmo propósito, que é a matriz de confusão, a partir da qual todas essas métricas podem ser derivadas.

Acurácia

A métrica de acurácia representa de uma forma geral o desempenho de um modelo, levando em conta o número de inferências corretas(c) que ele teve em relação ao número total de inferências(t):

$$\text{Acurácia} = \frac{c}{t}$$

No geral, ela só é efetiva quando o conjunto de dados é muito bem balanceado, ou seja, quando o número de exemplos pertencentes à categoria positiva é quase igual ao total de exemplos negativos. No caso em que ele é desbalanceado, mesmo tendo uma boa acurácia, pode ser que o modelo teve um bom desempenho apenas para uma das categorias de classificação, sendo que errou muito nas inferências dos exemplos da outra categoria.

Precisão e Recall

Nesse contexto de classificação binária, a precisão e a *recall* são métricas mais efetivas do que a acurácia, pois observam o resultado de uma forma mais granular.

A precisão verifica a quantidade de exemplos corretamente classificados como positivos em relação ao total de exemplos que o modelo inferiu como sendo positivos, fornecendo um entendimento a respeito da quantia de falsos positivos que o modelo entregou. Sendo tp o número de exemplos corretamente classificados como positivos, e fp a quantidade de falsos positivos, então a fórmula da precisão é dada por:

$$\text{Precisão} = \frac{tp}{tp+fp} \text{ (Jurafsky e Martin, 2020)}$$

Já a *recall* observa a quantidade de exemplos corretamente classificados como positivos em relação à quantidade total de exemplos realmente positivos, provendo, assim, uma compreensão melhor no que se refere ao total de falsos negativos. Sendo tp a quantidade de exemplos corretamente classificados como positivos, e fn a quantidade de falsos negativos, então a fórmula da *recall* é dada por:

$$\text{Recall} = \frac{tp}{tp+fn} \text{ (Jurafsky e Martin, 2020)}$$

A métrica F_1, ou F_1-Score, vem de uma média harmônica(Weisstein, 2024) entre a precisão e a *recall*. Ela proporciona um balanceamento entre essas duas métricas. Dados os

valores da precisão, P, e da *recall*, R, sua fórmula é:

$$F_1 = \frac{2 \cdot P \cdot R}{P+R}$$ (Jurafsky e Martin, 2020)

Além disso, ela penaliza bastante os casos em que uma das métricas é significativamente menor que a outra. Isso faz com que a F_1 seja uma boa escolha para detectar como o desempenho do modelo está considerando ambas as categorias.

Matriz de confusão

Numa matriz de confusão, é possível obter todos os valores necessários para os cálculos das métricas mencionadas anteriormente. Ela é uma tabela para se visualizar como um modelo desempenhou em relação a um conjunto de dados, usando duas dimensões: resposta do modelo e real categoria do dado, ambas podendo assumir os valores positivo e negativo (Jurafsky e Martin, 2020).

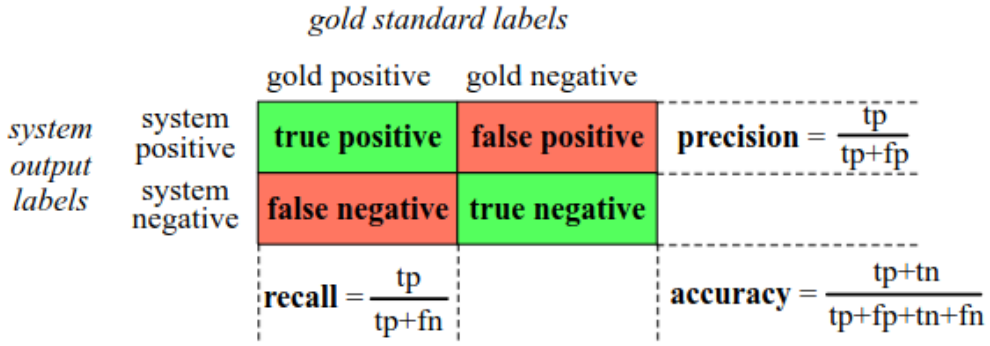


Figura 2.4: *Exemplo de matriz de confusão*

Na figura 2.4 (Jurafsky e Martin, 2020) acima, há um exemplo de matriz de confusão o qual também servirá como referência. No seu eixo y, está a inferência do modelo, onde o valor inferior é a classificação negativa e o superior a positiva. Já no eixo x, está a real categoria do dado, onde o valor à esquerda é a classificação positiva e à direita a negativa.

Dessa forma, no canto superior esquerdo estarão os exemplos corretamente classificados como positivos, no canto superior direito, os falsos positivos, no canto inferior esquerdo os falsos negativos, e, por fim, no canto inferior direito os verdadeiros negativos.

2.3 Ferramentas de programação

Por fim, será feita uma breve contextualização a respeito das ferramentas de programação utilizadas para a aplicação dos conceitos teóricos apresentados anteriormente, dentre elas: a linguagem de programação *Python*, a plataforma *Jupyter Notebook* e, por fim, a biblioteca *Hugging Face*.

2.3.1 Python

Python é uma linguagem de programação de alto nível conhecida por sua simplicidade e legibilidade. Devido à ênfase dada à legibilidade do código e facilidade de uso ao longo

de seu desenvolvimento, ela se tornou uma linguagem indispensável para uma extensa gama de contextos, tanto acadêmicos quanto no mercado de trabalho.

No contexto de inteligência artificial, e, mais especificamente, de aprendizado de máquina, ela é uma das linguagens mais usadas para o desenvolvimento de aplicações desse tipo, oferecendo um ecossistema muito rico de bibliotecas e *frameworks* voltados para soluções para essa área, como *PyTorch*, *TensorFlow*, *scikit-learn* e *Keras*.

Tendo em vista todos esses aspectos, e também a experiência do autor com essa linguagem no âmbito do mercado de trabalho, ela foi a linguagem escolhida para a implementação dos experimentos realizados ao longo deste trabalho.

Dessa forma, a fim de se obter uma melhor compreensão do desenvolvimento deste trabalho, serão listados aqui os conceitos principais dessa linguagem que foram utilizados nele, juntamente com uma referência para estudo da documentação oficial da linguagem(Foundation, 2024) caso necessário:

- Tipos de dados: https://docs.python.org/3.12/reference/datamodel.html#data-model
- Variáveis: https://docs.python.org/3.12/reference/expressions.html#atoms
- Sentenças condicionais (comando `if`): https://docs.python.org/3.12/reference/compound_stmts.html#the-if-statement
- Laços (comandos `for` e `while`): https://docs.python.org/3.12/reference/compound_stmts.html#the-for-statement
- Funções: https://docs.python.org/3.12/reference/compound_stmts.html#function-definitions
- Programação defensiva (comandos `try` e `except`): https://docs.python.org/3.12/reference/compound_stmts.html#the-try-statement
- Módulos e pacotes: https://docs.python.org/3.12/reference/import.#packages
- Classes e objetos: https://docs.python.org/3/tutorial/classes.html

2.3.2 *Jupyter Notebooks*

Um *Jupyter notebook*(website do projeto *Jupyter*) é uma plataforma interativa que permite a criação e compartilhamento de código, equações, visualizações e texto. Ela é muito útil dentro do contexto de aprendizado de máquina e ciência de dados, pois permite executar qualquer código em células, possibilitando, assim, uma experimentação rápida, trazendo os resultados da execução de uma forma amigável ao usuário.

Além disso, o suporte do *Jupyter* para visualizações possibilita a inspeção imediata de variáveis, conjuntos de dados, do progresso de um *pipeline* de dados (e.g., o treinamento de um modelo em uma barra de progresso), gráficos, dentre outros elementos importantes a um experimento.

Ademais, esses *notebooks* funcionam como ferramentas eficazes de comunicação, permitindo a inclusão de texto, equações matemáticas e imagens dentro do mesmo documento.

Isso traz diversos benefícios ao apresentar e documentar fluxos de trabalho de aprendizado de máquina, hipóteses e descobertas.

A possibilidade de exportar *notebooks* para diversos formatos, como HTML ou PDF, facilita a criação de relatórios abrangentes e compartilháveis, aprimorando a reprodutibilidade e transparência de experimentos de aprendizado de máquina num ambiente colaborativo.

Por fim, vale dizer que a possibilidade de executar esses *notebooks* num ambiente da cloud(Google Colab), agilizou de forma extraordinária a execução de todas as etapas dos experimentos, devido ao uso de recursos computacionais modernos.

Por conta de todos esses benefícios, os *Jupyter Notebooks* foram utilizados durante todo o desenvolvimento dos experimentos presentes neste trabalho, tanto para as etapas de preprocessamento de dados quanto as de treinamento e validação dos modelos.

2.3.3 *Hugging Face*

A biblioteca *Hugging Face* tem como finalidade o desenvolvimento de modelos de aprendizado de máquina, fornecendo diversas ferramentas necessárias em todas as etapas do processo de elaboração de um modelo. Ela foi construída em cima de outras bibliotecas de aprendizado de máquina(e.g., *PyTorch* e *TensorFlow*), as quais possuem detalhes de implementação de baixo nível.

Dessa forma, ela traz uma maior abstração para esse processo de desenvolvimento, provendo funcionalidades prontas como, por exemplo:

- `Dataset`: representação de conjuntos de dados, incluindo diversas operações que podem ser aplicadas sobre eles.
- `Tokenizer`: conceito de uma ferramenta que pode ser usada para dividir um texto em unidades atômicas chamadas de *tokens*, os quais podem ser palavras, pedaços de palavras ou até letras. O objetivo da *tokenização* é preprocessar e representar o texto num formato que pode ser mais facilmente processado pelo modelo.
- `Model`: abstração para o modelo de aprendizado, sendo que é possível a instanciação de modelos pré-treinados a fim de facilitar o desenvolvimento de novos modelos por meio da estratégia de *fine-tuning*.

Pelo fato de essa biblioteca já provisionar diversas abstrações prontas, e também vários modelos pré-treinados, ela foi escolhida para a implementação dos experimentos a fim de que o foco do trabalho fosse apenas as tarefas específicas às quais foi proposto, ou seja, a detecção de fuga ao tema e de aglomerados de palavras.

Dessa forma, por exemplo, não foi necessário se preocupar em treinar um modelo que pudesse representar palavras em português como *embeddings*. Utilizou-se um modelo pré-treinado para essa tarefa, e realizou-se um refinamento nele para que fosse capaz de desempenhar bem numa nova tarefa(nesse caso, a detecção de fuga ao tema).

Capítulo 3

Datasets

Em todos os experimentos, foram utilizados três *datasets* ao total, sendo todos provenientes do trabalho Silveira *et al.*, 2024, o qual foi aceito na 16a Conferência Internacional em Processamento Computacional de Português (PROPOR 2024). Nele, há dois *datasets* de redações, *source A* (menor) e *source B* (maior), e um *dataset* de textos motivadores, no qual estão descritos os temas. Para facilitar as explicações seguintes, o *dataset* pequeno de redações será chamado de *A*, o grande de *B*, e o de temas de *C*.

3.1 Datasets de redações

O dataset *B* possui dados de 3219 redações, enquanto que o dataset *A* possui dados de apenas 387. Ambos possuem um *schema* muito semelhante, sendo que as colunas do maior são um subconjunto das colunas do menor. Assim, explicando o *schema* do *A*, ter-se-á explicado o schema do *B*.

Assim, vale apresentar a estrutura do *dataset A*. Ele possui 10 colunas:

- `"id"`: um número inteiro que identifica a redação;
- `"id_prompt"`: um número inteiro que identifica o tema proposto para a redação;
- `"title"`: uma *string* que contém o título dado à redação;
- `"essay"`: uma *string* com o texto da redação dividido em parágrafos;
- `"grades"`: uma lista de inteiros com as notas recebidas pela redação em questão para cada competência avalida no ENEM;
- `"final_grade"`: um inteiro que representa a nota final da redação, ou seja, a soma de todas as notas presentes na lista grades;
- `"is_ENEM"`: um booleano que indica se a redação foi elaborada durante a realização do ENEM ou não;
- `"is_convertible"`: não foi possível compreender a finalidade desse campo;
- `"general"`: uma *string* com comentários gerais sobre o texto, elaborados pelo corretor;

- `"specific"`: uma *string* com comentários elaborados pelo corretor os quais são específicos para cada uma das competências;

As únicas colunas que não estão presentes no dataset *B* são: is_ENEM e is_convertible.

3.2 Dataset de temas

Já o dataset de temas, *C*, tem um pequeno número de linhas(44), e os textos motivadores e os temas propostos em cada uma das suas linhas estão relacionados com as redações do dataset de redações pequeno. Suas colunas são:

- `"id"`: um número inteiro que identifica o par tema e texto motivador;
- `"source"`: uma string que indica de onde foram obtidos os dados da linha em questão, e.g., UOL;
- `"title"`: uma string que contém o título dado ao tema;
- `"prompt"`: uma string com um texto que explica o que se espera que a redação do estudante responda;
- `"supporting_text"`: uma string que contém os textos motivadores do tema em questão;

Capítulo 4

Desenvolvimento

Tendo em vista a fundamentação teórica, e a explicação a respeito das ferramentas e dos datasets utilizados neste projeto, agora será exposto como se deu o seu desenvolvimento. Foram realizados experimentos para duas tarefas principais: a detecção de fuga ao tema e a detecção de aglomerado de palavras em redações do ENEM. As especificidades da hipótese relacionada à cada tarefa, do tratamento dos dados, da implementação e dos resultados de cada um dos experimentos serão abordadas a seguir de forma separada.

4.1 Detecção de fuga ao tema

4.1.1 Hipótese

Como foi explicado na seção 2.1, a fuga ao tema é uma situação que leva uma redação a receber nota 0 em relação à competência II. Dessa forma, nos experimentos realizados em relação a essa tarefa, buscou-se implementar um sistema que recebe dois textos como entrada, o texto da redação e o texto que contém a proposta temática, e fornece uma saída que é uma predição a respeito de essa redação estar fugindo desse tema ou não.

Assim, a fim de explorar as nuances dessa tarefa, utilizou-se inicialmente um *Jupyter Notebook* para se implementar todos os recursos necessários, devido a maior liberdade e rapidez que essa ferramenta fornece em tarefas de estudo e investigação. Esse *notebook* se encontra no caminho *correction/topic_deviation/experiment.ipynb* do repositório do projeto (link).

4.1.2 Pré-processamento dos dados

Os *datasets* utilizados nesse experimento foram o *dataset A* (vide 3), presente no arquivo *correction/data/raw/redacoes.csv* e o *dataset C* (vide 3), presente no arquivo *correction/data/-raw/temas.csv*. No início, utilizou-se as bibliotecas *Hugging Face* e *Pandas* para se explorar e compreender melhor esses dois *datasets*.

Em seguida, percebeu-se que era necessário realizar um pré-processamento em ambos os *datasets* para que eles pudessem ser finalmente usados nos experimentos. Todos os

pré-processamentos foram implementados inicialmente no *notebook*. Depois, foi feita uma refatoração no código, e a lógica de pré-processamento específica de cada *dataset* ficou centralizada em uma classe distinta.

Pré-processamento das redações

Para esse *dataset* de redações, toda a lógica de preprocessamento foi estruturada na classe `EssayDatasetPreprocessorForTopicDeviation`, presente no arquivo *correction/topic_deviation/essay_preprocessor.py*.

Na primeira parte dessa lógica, foram renomeadas algumas colunas, a fim de que elas tivessem nomes mais autoexplicativos. Para isso, utilizou-se o mapeamento presente no seguinte dicionário da linguagem *Python*:

Programa 4.1 Dicionário para renomear colunas do *dataset* de redações.

```
1   RENAME_COLUMNS_MAP = {
2      "id": "essay_id",
3      "id_prompt": "topic_id",
4      "essay": "essay_text",
5   }
```

As chaves desse dicionário são os nomes iniciais das colunas, e o valor de cada chave é o novo nome que a coluna irá assumir. Dessa forma, a coluna `id` foi renomeada para `essay_id`, a coluna `id_prompt` para `topic_id` e a coluna `essay` para `essay_text`, e o método utilizado para fazer essa renomeação foi o `Dataset.rename_columns` da biblioteca *Hugging Face*.

Em seguida, foi necessário preprocessar o texto da redação. No dado cru, isto é, no *dataset* inicial, cada redação era uma lista de *strings*, em que cada *string* era um parágrafo. A fim de que se tivesse apenas uma *string* com todos os caracteres da redação, foi executado o método apresentado no programa 4.2 abaixo:

Programa 4.2 Método para pré-processar o texto da redação

```
1   def _preprocess_essay(self, example):
2      essay_before = example["essay_text"]
3      essay_after = ""
4
5      paragraphs = eval(essay_before)
6      for paragraph in paragraphs:
7         essay_after += paragraph
8
9      example["essay_text"] = essay_after
10     return example
```

Esse método itera sobre os parágrafos da redação(lista `paragraphs`), e concatena um parágrafo ao próximo utilizando o operador *+=*, obtendo assim, no final, o texto completo da redação.

O próximo passo foi criar uma coluna chamada `label`, a qual contém um rótulo que diz se uma redação fugiu ou não do tema proposto. Como já foi explicado na seção 2.1, as redações que fogem do tema recebem nota 0 na competência II.

Programa 4.3 Método para criar coluna do rótulo de fuga ao tema

```
1   def _create_column_for_topic_deviation_label(self, example):
2      grades_list = eval(example["grades"])
3      second_grade = grades_list[self.SECOND_GRADE_INDEX]
4
5      if second_grade > self.GRADE_THRESHOLD:
6         example["label"] = self.POSITIVE_LABEL
7      else:
8         example["label"] = self.NEGATIVE_LABEL
9
10     return example
```

Como se pode observar no método `_create_column_for_topic_deviation_label` presente no programa 4.3, a fim de criar esse rótulo, partiu-se da coluna *grades*, a qual continha a nota para cada competência, sendo a primeira nota a da competência I, a segunda nota a da competência II e assim por diante. O atributo `SECOND_GRADE_INDEX` da classe `EssayDatasetPreprocessorForTopicDeviation` é uma constante com o valor 1, que é o índice do elemento da lista de notas que contém a nota da competência II. Portanto, acessando a lista de notas utilizando esse índice, a nota da segunda competência foi armazenada na variável `second_grade`.

Agora, a partir dessa nota, é possível dar um rótulo de fuga ao tema para a redação em questão. O outro atributo `GRADE_THRESHOLD` dessa classe também é uma constante e possui o valor 0. Assim, pela sentença condicional presente nas linhas 5-8 desse programa, todas as redações cuja nota foi maior do que 0, ou seja, aquelas que abordaram o tema, possuirão o rótulo `self.POSITIVE_LABEL` (cujo valor é 1), e aquelas cuja nota foi 0, ou seja, as que fugiram do tema, o rótulo `self.NEGATIVE_LABEL` (cujo valor é 0).

Em seguida, percebeu-se um ponto importante: os dados obtidos estavam completamente desbalanceados. O *dataset* em questão possui 385 redações, sendo que apenas 57 delas fogem ao tema. Isso dificultaria muito o treinamento de um modelo para detectar a fuga ao tema, pois ele poderia ficar enviesado para redações que não fogem ao tema devido à diferença da quantidade de dados das duas categorias.

Tendo isso em vista, surgiu uma ideia para contornar essa situação: criar dados artificiais para o treinamento. Como o objetivo é ter um modelo que prediz se uma redação foge ou não do tema proposto, sendo que se conhece a informação de qual tema foi proposto a cada redação, é possível combinar redações que não fugiram do seu tema com algum tema diferente do que foi proposto a ela. Dessa forma, pode-se entregar esse dado artificial ao modelo com um rótulo de fuga ao tema a fim de que ele aprenda que de fato essa redação não tem relação alguma de abordagem com esse tema.

Logo, a etapa posterior do preprocessamento foi a execução dessa ideia de aumento de dados, expressa em código no programa 4.4 abaixo. A fim de deixar o *dataset* balanceado, ou seja, tendo o mesmo número de exemplos positivos e negativos, foram criados 380 exemplos

artificiais, número expresso pela variável `number_of_artificial_examples` na linha 13. Como se pode observar nas linhas 17-19, são escolhidas redações que não fugiram ao tema de forma aleatória utilizando o método `sample` da classe `pandas.DataFrame`(documentação dessa classe). Na sequência, finaliza-se essa etapa combinando de fato essas redações aleatórias com um tema que não fosse o seu com o auxílio do método `_random_integer_with_blacklist`. Ele escolhe um tema aleatório dentre uma lista de temas, excluindo sempre a possibilidade de que o tema seja o mesmo tema proposta para essa redação.

Programa 4.4 Métodos para criar dados artificiais

```
1   def _create_artificial_negative_examples_dataset(self):
2       positive_examples_dataset = self.train_dataset.filter(
3           lambda example: example["label"] == self.POSITIVE_LABEL
4       )
5       negative_examples_dataset = self.train_dataset.filter(
6           lambda example: example["label"] == self.NEGATIVE_LABEL
7       )
8
9       positive_examples_dataset.set_format("pandas")
10
11      positive_examples_df = positive_examples_dataset[:]
12
13      number_of_artificial_examples = len(positive_examples_dataset) - len(
14          negative_examples_dataset
15      )
16
17      artificial_negative_examples_df = positive_examples_df.sample(
18          n=number_of_artificial_examples
19      )
20
21      min_topic_id = min(self.train_dataset["topic_id"])
22      max_topic_id = max(self.train_dataset["topic_id"])
23
24      artificial_negative_examples_df["topic_id"] =
            artificial_negative_examples_df[
25          "topic_id"
26      ].apply(
27          lambda topic_id: self._random_integer_with_blacklist(
28              min_topic_id, max_topic_id, [topic_id]
29          )
30      )
31      artificial_negative_examples_df["label"] = self.NEGATIVE_LABEL
32
33      artificial_negative_examples_dataset = Dataset.from_pandas(
34          artificial_negative_examples_df, preserve_index=False
35      )
36
37      return artificial_negative_examples_dataset
38
39   def _random_integer_with_blacklist(self, min_val, max_val, blacklist):
40      while True:
41          random_num = random.randint(min_val, max_val)
42          if random_num not in blacklist:
```

cont ⟶

⟶ *cont*

```
43              return random_num
```

Após essa importante etapa, chegou o momento de remover algumas colunas que não serão mais utilizadas. São elas as strings presentes na lista `COLUMNS_TO_REMOVE` que é um atributo da classe de preprocessamento, presentes no programa 4.5 abaixo. Elas foram removidas com o auxílio do método `Dataset.remove_columns` da biblioteca *Hugging Face*.

Programa 4.5 Colunas a serem removidas do *dataset* de redações

```
1  COLUMNS_TO_REMOVE = [
2     "title",
3     "final_grade",
4     "is_ENEM",
5     "is_convertible",
6     "general",
7     "specific",
8     "grades",
9  ]
```

Uma prática importante quando se trata do treinamento de um modelo de aprendizado de máquina é a divisão do *dataset* em três subconjuntos: o de treinamento, o de teste e o de validação. Já que todas as transformações necessárias e convenientes foram feitas sobre o *dataset* inicial, precisa-se dividir esse *dataset* resultante nessas três partes, sendo que 70% dos seus dados serão destinados ao conjunto de treinamento, 15% para o de teste e 15% para o de validação.

Porém, não foi feita uma divisão aleatória entre os exemplos de teste e validação. No *dataset* de teste, por ser usado como uma forma de validação final do modelo num contexto real de correção, foram inseridos apenas exemplos reais de ambas as categorias de dados, ou seja, a categoria das redações que fugiram do tema, e a categoria daquelas que abordaram o tema. A criação do *dataset* de teste nesses moldes pode ser observada no método `_create_test_dataset` da classe de preprocessamento, presente no programa 4.6.

Nela, o primeiro passo é obter apenas os exemplos reais filtrando-os por meio da coluna `is_artificial`, separando-os em dois *datasets*. Na variável `original_negative_examples` são armazenados os exemplos de redações que fugiram ao tema, e na `original_positive_examples` , as que abordaram o tema. Depois, obtém-se o número total de exemplos que estarão no *dataset* de teste. Tal número é obtido na linha 11, multiplicando-se o tamanho do *dataset* original pela porcentagem definida para o *dataset* de teste (variável `TEST_DATASET_SIZE`), a qual era 15%, resultando em 98 exemplos.

Em seguida, a fim de se alcançar um *dataset* de teste equilibrado, calculou-se um mesmo número de exemplos negativos e positivos, ou seja, cada um resultou na metade do número total anteriormente calculado (49). Assim, a partir desses dois números, fez-se uma amostragem aleatória em ambos os datasets utilizando os métodos `Dataset.shuffle` para embaralhar os exemplos de uma maneira aleatória e `Dataset.select` para selecionar

os primeiros 49 exemplos de cada dataset embaralhado. Por fim, concatena-se um *dataset* ao outro, resultando no *dataset* de teste desejado.

Programa 4.6 Método para a criação do *dataset* de teste

```
1   def _create_test_dataset(self):
2       original_negative_examples = self.train_dataset.filter(
3           lambda example: example["label"] == self.NEGATIVE_LABEL
4           and example["is_artificial"] == False
5       )
6       original_positive_examples = self.train_dataset.filter(
7           lambda example: example["label"] == self.POSITIVE_LABEL
8           and example["is_artificial"] == False
9       )
10
11      num_of_examples = int(self.train_dataset_original_size * self.
            TEST_DATASET_SIZE)
12
13      num_of_negative_examples = int(math.floor(num_of_examples / 2))
14      num_of_positive_examples = num_of_negative_examples
15
16      test_negative_examples = original_negative_examples.shuffle().select(
17          range(num_of_negative_examples)
18      )
19
20      test_positive_examples = original_positive_examples.shuffle().select(
21          range(num_of_positive_examples)
22      )
23
24      test_dataset = concatenate_datasets(
25          [
26              test_positive_examples,
27              test_negative_examples,
28          ]
29      ).shuffle()
30
31      return test_dataset
```

De maneira análoga, finalizando o preprocessamento do *dataset* de redações, construiu-se o *dataset* de validação equilibrando exemplos positivos e negativos, porém não foram filtrados apenas exemplos reais, ou seja, o *dataset* de validação possui tanto exemplos reais quanto artificiais.

Pré-processamento dos temas

Agora, explicar-se-á o preprocessamento do *dataset* de temas, o qual foi encapsulado na classe `PromptDatasetPreprocessorForTopicDeviation` presente no arquivo `correction/topic_deviation/prompt_preprocessor.py`.

De maneira semelhante ao *dataset* de redações, o primeiro passo aqui é renomear a coluna `id` para `topic_id` utilizando o método `Dataset.rename_columns` da biblioteca *Hugging Face*, a fim de que, futuramente, se possa distinguir o identificador da redação e o do tema.

Em seguida, há o passo mais importante deste preprocessamento, que é a concatenação de três textos presentes nesse *dataset*: o título, o enunciado(ou *prompt*) e o texto motivador. Tal processo está expresso no método `_concatenate_texts` da classe de preprocessamento. Esse método, o qual está presente no programa 4.7 a seguir, inicia-se definindo a variável `concatenated_texts`, onde ficará armazenado o texto concatenado, e inicializando-a com uma *string* vazia. Depois, armazena-se os textos do enunciado, do texto motivador e do título, respectivamente nas variáveis `topic`, `supporting_text` e `title`.

Programa 4.7 Método para concatenar textos do tema

```
1   def _concatenate_texts(self, example):
2       concatenated_texts = ""
3
4       topic = example["prompt"]
5       supporting_text = example["supporting_text"]
6
7       # This try-catch block is necessary because some titles have quotes around
            it,
8       # but others don't.
9       try:
10          title = eval(example["title"])
11      except SyntaxError:
12          title = example["title"]
13
14      paragraphs = [title] + eval(supporting_text) + eval(topic)
15
16      for paragraph in paragraphs:
17          has_desired_ending = False
18          for ending in self.PARAGRAPH_ENDINGS:
19              if paragraph.endswith(ending):
20                  has_desired_ending = True
21                  break
22
23          if not has_desired_ending:
24              paragraph = paragraph + "."
25
26          concatenated_texts += paragraph
27
28      example["topic_text"] = concatenated_texts
29      return example
```

Como se pode ler nas linhas 7-12, para obter o texto do título, foi necessária uma estratégia de programação defensiva, que é o bloco `try`/`except`, pois nem todos os títulos estavam no mesmo formato. Alguns deles estavam com aspas ao redor, mas outros não. Assim, no bloco `try`, faz-se uma tentativa de obter o título removendo as aspas da *string* `example["title"]` aplicando nela a função *built-in* `eval` da linguagem Python. Caso as aspas não estejam presentes na *string* `example["title"]`, é lançada a exceção `SyntaxError`. Dessa forma, caso ela aconteça, o fluxo de execução do código vai para o bloco `except` na linha 11, no qual apenas se armazena essa mesma *string* sem remover nenhum caracter dela, pois já está pronta.

Na sequência, na linha 14, é construída uma lista com todas as *strings* que serão

concatenadas e ela é armazenada na variável `paragraphs`. Foi necessário aplicar novamente a função `eval` nos textos `supporting_text` e `topic` pois eles são *strings* cujo conteúdo é um texto no formato de uma lista da linguagem Python, como, por exemplo, `"['a', 'b', 'c']"`.

Tendo construído a lista `paragraphs`, faz-se uma iteração sobre cada parágrafo dentro dela. Para cada um deles, verifica-se se são terminados por alguma pontuação de final de frase, as quais estão presentes no atributo `PARAGRAPH_ENDINGS` dessa classe. São elas: o ponto final, o de exclamação e o de interrogação. Caso algum parágrafo não seja terminado por uma dessas pontuações, adiciona-se um ponto final ao fim dele, como está expresso na sentença condicional nas linhas 23 e 24. Após isso, o parágrafo em questão está finalmente pronto para ser concatenado, e a concatenação é então realizada, adicionando-o ao final da *string* `concatenated_texts`. Dessa forma, ao findar todas as iterações, essa *string* conterá todos os textos desejados de forma concatenada.

Combinação dos *datasets* preprocessados

O último passo do preprocessamento de dados para este experimento é combinar os dados do *dataset* de redações com os dados do *dataset* de temas. Tal combinação foi implementada na classe `EssayAndPromptDatasetsCombinator`, presente no arquivo `correction/topic_deviation/combinator.py`.

A lógica dessa combinação está presente no método `combine` dessa classe, como se pode observar no programa 4.8. Ele recebe como parâmetro um dicionário de *datasets* de redações(`essay_dataset_dict`) e um *dataset* de temas(`topic_dataset`). Nesse dicionário, as chaves são o nome do *split* do *dataset* em questão, e os valores são os próprios datasets. O nome do *dataset* pode ser um dos valores presentes no atributo `SPLITS` dessa classe, indicando se ele é um *dataset* de treinamento, teste ou validação.

Programa 4.8 Classe que combina os *datasets* de redações e de temas

```
1   class EssayAndPromptDatasetsCombinator:
2       COLUMNS_ORDER = ["essay_id", "essay_text", "topic_id", "topic_text", "
            label"]
3       SPLITS = ["test", "train", "validation"]
4
5       def combine(self, essay_dataset_dict: DatasetDict, topic_dataset: Dataset)
             -> DatasetDict:
6           topic_dataset.set_format("pandas")
7           topic_df = topic_dataset[:].set_index("topic_id")
8           combined_dataset_dict = {}
9
10          for split in self.SPLITS:
11              essay_dataset = essay_dataset_dict[split]
12              essay_dataset.set_format("pandas")
13
14              essay_df = essay_dataset[:]
15
16              combined_df = essay_df.join(topic_df, on="topic_id")
17              combined_df = combined_df.loc[:, self.COLUMNS_ORDER]
18
```

cont ⟶

⟶ *cont*

```
19            essay_dataset.set_format()
20
21            combined_dataset = Dataset.from_pandas(combined_df, preserve_index=
                  False)
22
23            combined_dataset_dict[split] = combined_dataset
24
25        combined_dataset_dict = DatasetDict(combined_dataset_dict)
26
27        return combined_dataset_dict
```

A fim de possibilitar a manipulação necessária, foi essencial transformar todos os *datasets* utilizados em objetos do tipo `pandas.DataFrame`. Como se pode observar nas linhas 6-7, por exemplo, tal transformação foi realizada por meio do método `Dataset.set_format` e do operador de *slicing* `[:]` selecionando todos os exemplos do *dataset*. De forma análoga, os *datasets* de redações também foram transformados em *dataframes* nas linhas 12-14.

A informação que permite juntar uma linha do *dataset* de redações com uma linha do *dataset* de temas é o identificador do tema, presenta na coluna *topic_id*. Assim, no *loop* presente nas linhas 10-23, foi feita essa combinação para cada *split* do *dataset* de redações. Na linha 16, utiliza-se o método `pandas.DataFrame.join`(documentação desse método) para criar um novo *dataframe*, `combined_df`, o qual terá uma linha para cada redação, e, agora, além dos dados da redação, possuirá também os dados do tema ao qual essa redação está relacionada. Depois, na linha 17, faz-se uma ordenação das colunas desse *dataframe* com o auxílio do método `pandas.DataFrame.loc` (documentação desse método). Por fim, é feita uma conversão desse *dataframe* para o tipo `Dataset` utilizando o método `Dataset.from_pandas` . Ao final da execução desse método, a variável `combined_dataset_dict` possuirá os três *datasets* (de treinamento, teste e validação), prontos para serem utilizados pelo modelo de detecção de fuga ao tema.

4.1.3 Treinamento do modelo

Tendo agora todos os dados necessários já preprocessados, pode-se dar seguimento ao treinamento de um modelo para realizar a tarefa de detecção de fuga ao tema. Todo o código desenvolvido que envolve o treinamento do modelo de detecção está localizado em duas classes, `TopicDeviationNeuralNetwork`, presente no arquivo *correction/topic_deviation/neural_network.py* e `TopicDeviationFineTuner`, a qual está no arquivo *correction/topic_deviation/fine_tuner.py*. O *notebook* onde essas classes foram de fato utilizadas para o treinamento, avaliação e teste do modelo é o *correction/topic_deviation/experiment.ipynb*

Classe TopicDeviationNeuralNetwork

A primeira classe representa a rede neural que foi construída e treinada nesse experimento. Ela está descrita no programa 4.9 abaixo. Como se pode observar na linha 7, ela herda da classe `nn.Module`, que é uma classe fundamental da biblioteca *PyTorch* a partir da qual se pode implementar redes neurais.

No seu método construtor, são instaciados dois objetos importantes: `self.model` e `self.classifier`. `self.model` é um modelo *transformer* pré-treinado proveniente do repositório de modelos *Hugging Face* o qual é identificado pelo seu `checkpoint`. O modelo pré-treinado escolhido para este experimento foi o BERTimbau Souza *et al.*, 2020, presente no checkpoint *neuralmind/bert-base-portuguese-cased*.

Programa 4.9 Classe da rede neural treinada para detecção de fuga ao tema

```
1   import torch
2   import torch.nn as nn
3   from torch.nn.functional import cross_entropy
4   from transformers import AutoModel
5
6
7   class TopicDeviationNeuralNetwork(nn.Module):
8      def __init__(self, checkpoint) -> None:
9         super(TopicDeviationNeuralNetwork, self).__init__()
10        self.model = AutoModel.from_pretrained(checkpoint)
11        self.classifier = nn.Sequential(nn.Linear(1536, 2), nn.Sigmoid())
12
13     def forward(self, essay_text, topic_text, labels=None):
14        outputs = self.model(**essay_text)
15        last_hidden_states = outputs.last_hidden_state
16        essay_representations = last_hidden_states[:, 0, :]
17
18        outputs = self.model(**topic_text)
19        last_hidden_states = outputs.last_hidden_state
20        topic_representations = last_hidden_states[:, 0, :]
21
22        joint_representations = torch.cat((essay_representations,
              topic_representations), dim=1)
23
24        logits = self.classifier(joint_representations)
25
26        training = labels is not None
27
28        if training:
29           loss = cross_entropy(logits, labels["labels"])
30           outputs = {"logits": logits, "loss": loss}
31        else:
32           outputs = {"logits": logits, "loss": None}
33
34        return outputs
```

Ele será o modelo que sofrerá um processo de refinamento para inferir corretamente a detecção de fuga ao tema. Já o `self.classifier` é um objeto da classe `nn.Sequential`, a qual serve para criar uma arquitetura sequencial de operações que serão executadas dentro da rede neural quando invocada. Nesse caso, ele primeiro aplica uma transformação linear sobre o *input* com o uso da classe `nn.Linear`, a qual transforma esse *input* de tamanho 1536 (que é o tamanho da representação do texto utilizado para o treinamento) em um *output* de tamanho 2. Esse *output* será o *input* para a próxima operação, que é a aplicação da função de ativação *sigmoid* (Jurafsky e Martin, 2020), representada pela classe `nn.Sigmoid`.

Em seguida, foi necessário implementar o método `forward`. Ele é definido pela superclasse `nn.Module`, e sua implementação é obrigatória em qualquer classe filha. Esse método representa o processo de passar um *input* para a rede neural e receber um *output* que tenha algum significado, nesse caso, o significado relacionado a fuga ao tema, e ele é invocado tanto durante o treinamento quanto no momento da inferência.

Na implementação desse método, presente no programa 4.9, há três etapas principais: converter os textos da redação e do tema em *embeddings*, juntar as suas representações numéricas e, aplicar as operações da rede aos dados de *input*.

Tal conversão é realizada com o auxílio do modelo BERT instanciado no construtor, `self.model`. Nas linhas 14-16 é realizada a conversão do texto da redação em *embeddings*, e em seguida, o texto do tema é convertido também nas linhas 18-20. Esses *embeddings* ficam então armazenados, respectivamente, nas variáveis `essay_representations` e `topic_representations`.

Agora, o segundo passo é juntar ambas essas representações numa única representação. A estratégia empregada para isso foi a utilização da função `torch.cat` para concatenar a representação do texto do tema ao final da representação da redação, como está descrito na linha 22. Como cada uma dessas representações possuía um tamanho de 768, o qual é o tamanho máximo de representações suportado pelo modelo `self.model`, a representação final ficou com tamanho $768 + 768 = 1536$, e foi armazenada na variável `joint_representations`.

Por fim, essa representação conjunta é passada pelo classificador `self.classifier` na linha 24, o qual retorna duas probabilidades, a probabilidade de a redação em questão ter fugido ao tema(categoria 0) e a probabilidade da mesma não ter fugido ao tema(categoria 1).

Como esse método pode ser executado tanto na etapa de treinamento quanto de inferência, ele precisa distinguir essas duas situações. Para isso, ele utiliza o parâmetro `labels`, que representa uma lista de rótulos(0 ou 1) que indicam qual a categoria à qual os exemplos de treinamento presentes nas variáveis `essay_text` e `topic_text` pertencem. Esse parâmetro só é utilizado na etapa de treinamento, pois serve como um gabarito para o modelo se basear a fim de melhorar o seu desempenho de inferência.

Assim, quando o valor desse parâmetro é `None` (valor padrão), pode-se observar, pela sentença presente na linha 26, que esse método infere que a rede está apenas num processo de inferência, pois a variável `training` recebe o valor `False`. Quando o valor dele é diferente de `None`, por exemplo, a lista `[1, 1, 0, 1]`, então, a variável `training` recebe o valor `True` e, portanto, o método infere que a rede está sendo treinada.

Dessa maneira, quando esse método infere que a rede está sendo treinada, há ainda um passo adicional, que é o cálculo da função de perda para o resultado alcançado. Na sentença condicional presente nas linhas 28-30, pode-se perceber que caso a variável `training` tenha o valor `True`, isto é, caso a rede esteja sendo treinada, aplica-se a função `cross_entropy`, a qual computa a perda entre o resultado (probabilidades) e os rótulos (Jurafsky e Martin, 2020).

Finalmente, as probabilidades calculadas e a perda são armazenadas na variável `outputs` como se vê nas linhas 30 e 32, e esta é retornada pela função `forward`.

Classe TopicDeviationFineTuner

Já a classe `TopicDeviationFineTuner` representa uma entidade que irá gerenciar todo o processo de treinamento e avaliação da rede neural, incluindo as etapas que os precedem, como a tokenização dos *inputs*, e as etapas que sucedem, como o cálculo e apresentação das métricas relativas à performance da rede neural nas fases de avaliação e teste.

Quando um objeto dessa classe é inicializado por meio do seu método construtor, são armazenados diversos atributos importantes que serão utilizados nas etapas posteriores, como se observa no programa 4.10 abaixo.

Programa 4.10 Método construtor do *fine-tuner*

```
1
2   def __init__(
3         self, checkpoint: str, datasets: DatasetDict, learning_rate: float = 5e
              -5, batch_size=4
4       ) -> None:
5         self.checkpoint = checkpoint
6         self.datasets = datasets
7         self.batch_size = batch_size
8
9         self.tokenizer = AutoTokenizer.from_pretrained(self.checkpoint,
              do_lower_case=False)
10        self.data_collator = DataCollatorWithPadding(tokenizer=self.tokenizer)
11        self.neural_network = TopicDeviationNeuralNetwork(self.checkpoint)
12        self.optimizer = AdamW(self.neural_network.parameters(), lr=
              learning_rate)
13
14        self.tokenized_essay_datasets = self.get_essay_tokenized_datasets()
15        self.tokenized_topic_datasets = self.get_topic_tokenized_datasets()
16
17        self.metrics = defaultdict(list)
18
19        self.split_to_results_by_epoch = {split: {} for split in self.splits}
```

A seguir, está uma breve explicação sobre cada um deles, afim de que as próximas etapas fiquem mais claras:

- `checkpoint`: é o identificador do modelo pré-treinado do repositório *Hugging Face* que será usado na classe da rede neural explicada na seção anterior.
- `datasets`: um dicionário que mapeia o nome dos conjuntos(`"train"`, `"test"`, `"validation"`) para o *dataset* preprocessado que será utilizado pela rede.
- `batch_size`: tamanho do *batch* de *inputs* que será passado para a rede durante o treinamento.
- `tokenizer`: objeto que será usado para transformar os *inputs* em *tokens* que o modelo consiga utilizar para treinar ou fazer as inferências.
- `data_collator`: responsável por realizar o *padding*(FACE, 2022) dos *inputs* tokenizados.

- `neural_network`: instância da rede neural `TopicDeviationNeuralNetwork` explicada na seção anterior.
- `optimizer`: aplica a otimização no processo de aprendizado utilizando o algoritmo Adam com decaimento de pesos Loshchilov e Hutter, 2019
- `tokenized_essay_datasets`: *tokens* dos *datasets* de redações divididos por *split*(treinamento, teste e validação)
- `tokenized_topic_datasets`: *tokens* dos *datasets* de tema divididos por *split*(treinamento, teste e validação)
- `metrics`: dicionário onde serão armazenadas as métricas computadas ao final de cada época de treinamento.
- `split_to_results_by_epoch`: dicionário onde serão armazenados os resultados de inferência nas fases de avaliação e teste, a fim de possibilitar a apresentação gráfica dos mesmos numa matriz de confusão.

Agora, tendo todas as ferramentas necessárias ao treinamento prontas para serem utilizadas, mostrar-se-á como ele foi implementado. O processo completo de treinamento foi encapsulado no método `run_model_training`, o qual está mostrado abaixo no programa 4.11.

No seu início(linhas 1-4), ele carrega todos os dados que serão utilizados em uma estrutura da biblioteca *PyTorch* que permite a iteração em *batches* chamada `DataLoader`. Assim, haverá um objeto dessa classe para os dados das redações, `essay_train_dataloader`, para os dados dos temas, `topic_train_dataloader`, e para os rótulos de detecção, `label_train_dataloader`.

Programa 4.11 Método de execução do treinamento do modelo de detecção de fuga ao tema

```
1   def run_model_training(self, num_epochs: int = 3):
2       essay_train_dataloader = self.get_data_loaders(self.
            tokenized_essay_datasets)["train"]
3       topic_train_dataloader = self.get_data_loaders(self.
            tokenized_topic_datasets)["train"]
4       label_train_dataloader = self.get_label_data_loader(self.datasets)["train"
            ]
5
6       num_training_steps = num_epochs * len(essay_train_dataloader)
7
8       self.learning_scheduler = get_scheduler(
9          "linear",
10         optimizer=self.optimizer,
11         num_warmup_steps=0,
12         num_training_steps=num_training_steps,
13      )
14
15      device = self.get_current_device()
16      self.set_neural_network_device(device)
```

cont ⟶

⟶ *cont*

```
17
18      self.train_progress_bar = tqdm(range(num_training_steps))
19
20      for epoch in range(num_epochs):
21          self.neural_network.train()
22
23          essays_iterator = iter(essay_train_dataloader)
24          topics_iterator = iter(topic_train_dataloader)
25          labels_iterator = iter(label_train_dataloader)
26
27          dataloaders = zip(essays_iterator, topics_iterator, labels_iterator)
28
29          accumulation_steps = 8
30
31          for essay_batch, topic_batch, label_batch in dataloaders:
32              essay_batch = {k: v.to(device) for k, v in essay_batch.items()}
33              topic_batch = {k: v.to(device) for k, v in topic_batch.items()}
34              label_batch = {k: v.to(device) for k, v in label_batch.items()}
35
36              outputs = self.neural_network(essay_batch, topic_batch, label_batch)
37
38              loss = outputs["loss"]
39              loss.backward()
40
41              # Perform optimization step after accumulation_steps batches
42              if self.train_progress_bar.n % accumulation_steps == 0:
43                  self.optimizer.step()
44                  self.learning_scheduler.step()
45                  self.optimizer.zero_grad()
46
47              self.train_progress_bar.update(1)
48
49          self.run_model_evaluation("validation", epoch)
50          self.run_model_evaluation("test", epoch)
```

A iteração em *batches* que essa estrutura permite é necessária pois ela ajuda a reduzir o uso de memória, que é muito alto durante o treinamento, aumentar o grau de paralelismo que pode ser empregado no processo, e também permitir o uso de hardwares modernos como a unidade de processamento gráfico(GPU) e a de tensores(TPU) para torná-lo mais eficiente em relação a todos os recursos utilizados.

Em seguida, na linha 6, deve-se obter o número total de iterações de treinamento para ser passado para o *scheduler* de treinamento. Esse número é obtido multiplicando o número de épocas pelo tamanho de um dos *data loaders*, já que o número de iterações em uma época é o número de batches que há em um dos *data loaders*, e então é armazenado na variável (`num_training_steps`).

Tendo agora esse número, pode-se instanciar o *scheduler* do treinamento, responsável por ajustar a taxa de aprendizado do otimizador durante a execução do treinamento, como é feito nas linhas 8-13.

Depois disso, é o momento de definir qual o dispositivo no qual a rede neural executará o seu treinamento, isto é, se ele será executado numa Unidade Central de Processamento comum (CPU) ou numa unidade de processamento mais robusta, como a GPU e a TPU, ambas acessáveis via a interface de programação de aplicação (API) CUDA(link). Essa escolha é realizada nas linhas 15 e 16.

Na chamada do método `get_current_device`, faz-se a escolha do dispositivo(`"cpu"` ou `"cuda"`), a qual é armazenada na variável `device`. Nessa escolha, é dada maior prioridade a uma dessas unidades mais robustas caso elas estejam presentes no hardware onde o treinamento está sendo executado. Se elas não estiverem presentes, o dispositivo escolhido será a própria CPU. Na sequência, por meio do método `set_neural_network_device`, a rede neural, incluindo todos os seus parâmetros e submódulos, será movida para o dispositivo especificado em `device`.

Tendo em vista os dados carregados, o *scheduler* e a definição do dispositivo de execução, pode-se iniciar a execução do treinamento, a qual está presente no laço das linhas 20-47, que é repetido uma vez para cada época. Assim, na linha 21, é chamado o método `train` da rede neural a fim de alterar o seu modo para o modo de treinamento.

Após isso, são preparados iteradores sobre os dataloaders definidos no método construtor utilizando a função `iter`, a qual recebe um objeto e retorna um iterador sobre os elementos desse objeto. No caso dos três dataloaders definidos nas linhas 23-25, em uma iteração do laço presente nas linhas 31-45, o elemento de cada um deles será um *batch* de dados. Dessa forma, a cada iteração, estará se lidando com três *batches*, um para os textos de redações(`essay_batch`), outro para os textos motivadores(`topic_batch`) e um último para os rótulos de treinamento(`label_batch`), os quais são definidos e movidos para o `device` mencionado anteriormente nas linhas 32-34.

Em seguida, dentro desse segundo laço, é o momento de realizar o *forward pass* da rede neural, ou seja, executar o método `forward` dessa rede, cujo funcionamento está explicado anteriormente nesta mesma seção, o que ocorre na linha 36, sendo que os resultados são armazenados na variável `outputs`.

Como já foi explicado a respeito desse método, no caso da execução de um treinamento, seus resultados são um dicionário contendo as probabilidades de que os exemplos pertençam a um dos dois rótulos possíveis(chave `"logits"`) e um *tensor* com a perda calculada com a estratégia de *Cross Entropy* (Jurafsky e Martin, 2020). Assim, nas linhas 38-39, armazena-se essa perda na variável `loss`, e em seguida é executado o algoritmo de *Backward Propagation*(Jurafsky e Martin, 2020) através do método `Tensor.backward` para computar o gradiente da perda em relação aos parâmetros atuais do modelo.

Depois de acumular os gradientes calculados nas últimas `accumulation_steps`(nesse caso, 8) iterações, o passo de otimização é finalmente realizado, usando o gradiente calculado para atualizar os parâmetros da rede, efetivamente minimizando a perda. Nesse contexto, foi necessária a implementação do acúmulo de gradientes devido a limitação de memória que se teve no momento da execução, pois essa estratégia permite a atualização menos frequente dos parâmetros da rede, a qual é bastante custosa em termos de memória, e não deixa de alcançar uma atualização relevante.

Por fim, ao final de cada época de treinamento, é executado, nas linhas 49 e 50, o

método `run_model_evaluation` para realizar os processos de avaliação e teste do modelo. O código desse método é muito parecido com o do método `run_model_training`, portanto, é importante salientar as suas diferenças:

- Uso do método `eval` no lugar do método `train` a fim de colocar a rede neural no modo de avaliação.
- Execução do *forward pass* dentro do *context manager* `torch.no_grad`, o qual desabilita o cálculo de gradiente já que ele não é necessário para a avaliação, e aproveita-se também para reduzir o uso de memória.
- Cálculo de métricas como acurácia e F_1(vide seção 2.2.4 e seu armazenamento no atributo `metrics`.
- Armazenamento dos resultados de inferência em cada época no atributo `split_to_results_by_epoch`. Eles consistem nas predições feitas pela rede juntamente com o rótulo esperado para o exemplo em questão, a fim de que seja possível construir matrizes de confusão usando tais dados.

Devido à grande demora na execução do treinamento numa máquina comum, optou-se por rodar todo o processo descrito anteriormente, desde o preprocessamento dos dados até o momento da avaliação, na plataforma *Google Colab*.

Nela, foi possível executar todo o código numa TPU, o que reduziu o tempo de execução drasticamente, pois na máquina do autor a previsão para a completude do processo era de muitas horas ou até dias, porém dentro dessa plataforma, esse tempo foi reduzido para aproximadamente 15 minutos.

Mesmo usando essa plataforma, houve alguns desafios relacionados ao uso de memória, o qual acabou excedendo os limites ao longo dos testes que foram feitos. Foram necessários diversos testes com diferentes parametrizações do número de *batches*, e também alguns ajustes de código (como, por exemplo, a acumulação de gradiente) para encontrar uma configuração na qual fosse possível iniciar e terminar todo o processo sem que nenhuma exceção fosse lançada por motivos de falta de recursos.

4.1.4 Resultados da avaliação e teste do modelo

Depois de executar diversos testes, o teste que trouxe o melhor resultado foi com a seguinte parametrização:

- Taxa de aprendizado (*learning rate*): $2 \cdot 10^{-5}$
- Número de *batches*: 3

Em todos os testes, o treinamento foi executado ao longo de 11 épocas. Abaixo, está a tabela 4.1, onde pode-se notar que, tanto para o *dataset* de validação quanto para o de teste, os melhores resultados para as métricas de acurácia e F1 foram a da última(décima primeira) época.

Época	Ac. Validação	Ac. Teste	F1 Validação	F1 Teste
0	0.5051	0.5	0.6711	0.6667
1	0.5051	0.5	0.6711	0.6667
2	0.5051	0.5	0.6711	0.6667
3	0.5051	0.5	0.6711	0.6667
4	0.5657	0.5	0.6993	0.6667
5	0.7879	0.7143	0.8205	0.7742
6	0.7677	0.7653	0.7850	0.7890
7	0.7879	0.7959	0.8073	0.8246
8	0.7778	0.7755	0.7885	0.7925
9	0.8283	0.8061	0.8468	0.8319
10	0.8283	0.8061	0.8468	0.8319

Tabela 4.1: *Acurácia e F_1 nos* datasets *de validação e teste*

A acurácia de aproximadamente 80% indica que, no geral, ou seja, sem se fazer uma distinção entre as categorias existentes, a rede neural teve uma taxa de acerto de 80%. Já o resultado de aproximadamene 84% para a métrica F_1 indica que o modelo alcançou um bom equilíbrio entre as métricas de precisão e recall(vide seção 2.2.4), onde ambas são relativamente altas. Tal equilíbrio aponta que o modelo está classificando corretamente tanto as instâncias da classe positiva quanto as da classe negativa, minimizando, assim, falsos positivos e negativos.

4.2 Modelo de detecção de aglomerado de palavras

4.2.1 Hipótese

Assim como para as redações que fogem ao tema proposto, a ocorrência de redações que são compostas por aglomerados de palavras também é bastante rara. Além disso, como mencionado anteriormente na seção 2.1, o aglomerado de palavras é um texto de difícil compreensão, devido ao fato de ser composto por palavras que não constroem um significado completo juntas.

Tendo a premissa de que o modelo escolhido para este experimento, no caso, o BERTimbau(*checkpoint neuralmind/bert-base-portuguese-cased*), foi pré-treinado com textos de boa qualidade, ou seja, que foram estruturados de forma que um leitor qualquer consiga compreendê-lo, a hipótese levantada para este experimento foi que a probabilidade de que as palavras de um aglomerado ocorram uma ao lado da outra é baixa. Portanto, a perplexidade(vide seção 2.2.3) desse tipo de texto deve ser alta, e o inverso também deveria valer, ou seja, se a perplexidade de um texto é alta, então ele deveria ser um aglomerado de palavras.

Da mesma forma que no experimento anterior, neste experimento, foi usado um *Jupyter Notebook* para a exploração dos dados, do modelo pré-treinado, e de possíveis estratégias para a validação desta hipótese. O *notebook* em questão está localizado no caminho *correction/cluster_of_words/experiment.ipynb* do repositório deste projeto (link).

4.2.2 Pré-processamento dos dados

Neste experimento, ambos os *datasets* de redações foram utilizados: o *dataset A* (*correction/data/raw/redacoes.csv*) e o *dataset B* (*correction/data/raw/redacoes_grande.csv*). Como já havia sido realizada uma exploração sobre o primeiro *dataset* no experimento anterior e como a estrutura de ambos é muito semelhante, não foi necessária uma exploração adicional.

Com isso, partiu-se diretamente para a implementação do pré-processamento desses dois *datasets*, porém, agora, tendo em vista a tarefa de identificação de aglomerados de palavras. De maneira análoga ao primeiro experimento, todo o processo, desde o pré-processamento até os testes finais deste sistema, foram implementados, num primeiro momento, dentro do *notebook*.

Posteriormente, foi realizada uma refatoração completa no código desse *notebook*, inserindo cada pedaço de código responsável por uma tarefa específica dentro de uma classe. Cada uma dessas classes foi colocada dentro de um módulo *Python* separado, para que fosse importado pelo *notebook*. Assim, ele ficaria mais abstraído, e estaria focado apenas na execução do experimento em si, sem haver muitos detalhes de implementação que não são necessários a um usuário qualquer que queria executar o experimento.

Dessa forma, a classe responsável por realizar o pré-processamento dos dados de ambos os *datasets* de redações é a `EssayDatasetPreprocessorForClusterOfWords`, presente no arquivo *correction/cluster_of_words/preprocessor.py*. No seu método construtor, descrito a seguir

no programa 4.12 ela recebe dois parâmetros: `dataset`, um objeto do tipo `datasets.Dataset` (documentação dessa classe), o qual contém um dos *datasets* de redação, e `columns_to_remove` , uma lista de *strings* que contém as colunas que devem ser removidas desse *dataset*. Como se pode observar nas linhas 2-3 desse programa, ambos os parâmetros recebidos são armazenados em atributos internos, `self.original_dataset` e `self.columns_to_remove` para uso posterior.

Programa 4.12 Método construtor do preprocessador de redações

```
1   def __init__(self, dataset: Dataset, columns_to_remove: list):
2       self.original_dataset = dataset
3       self.columns_to_remove = columns_to_remove
```

Depois de um objeto do tipo dessa classe ser construído chamando esse método, é o momento de realizar de fato o pré-processamento. Para isso, foi implementado um outro método, chamado `_preprocess_dataset`, presente no programa 4.13 abaixo.

Programa 4.13 Método _preprocess_dataset

```
1   def preprocess_dataset(self) -> DatasetDict:
2       self.train_dataset = (
3           self.original_dataset.rename_columns(self.RENAME_COLUMNS_MAP)
4           .map(self._preprocess_essay)
5           .map(self._create_column_for_second_grade)
6       )
7
8       self.train_dataset = self.train_dataset.remove_columns(self.
            columns_to_remove)
9
10      self.dataset_dict = DatasetDict({"train": self.train_dataset})
11
12      return self.dataset_dict
```

Nesse método, há quatro passos principais:

1. Renomear as colunas do *dataset* com nomes mais mnemônicos
2. Preprocessar cada uma das redações
3. Criar uma coluna que contenha a nota da redação na competência II
4. Remover colunas que não serão necessárias

A renomeação das colunas do *dataset* é realizada na linha 3 através do método `rename_columns`, o qual utiliza o mapeamento presente no dicionário `self.RENAME_COLUMNS_MAP`, que tem o seguinte valor:

Programa 4.14 Mapeamento para renomeação das colunas do *dataset* de redações

```
1  RENAME_COLUMNS_MAP = {
2     "id": "essay_id",
3     "essay": "essay_text",
4  }
```

Em seguida, na linha 4, cada uma das redações será preprocessada de acordo com a mesma lógica presente no método `_preprocess_essay` da classe `EssayDatasetPreprocessorForTopicDeviation`, explicada na seção 4.1.2.

Já para aplicar o terceiro passo mencionado na linha 5, foi necessária a aplicação de um método auxiliar chamado `_create_column_for_second_grade`, mostrado no programa 4.15 a seguir. Seu funcionamento é semelhante ao do método `create_column_for_topic_deviation_label` da classe `EssayDatasetPreprocessorForTopicDeviation`, porém, sem a necessidade de se calcular um rótulo, pois o dado de interesse aqui é de fato qual foi a nota da redação para a competência II.

Programa 4.15 Método que cria coluna com a nota da segunda competência

```
1  def _create_column_for_second_grade(self, example):
2     grades_list = eval(example["grades"])
3     second_grade = grades_list[self.SECOND_GRADE_INDEX]
4     example["second_grade"] = second_grade
5     return example
```

Assim, na linha 2 deste programa, se obtém a lista de notas para todas as competências, a qual fica armazenada na variável `grades_list`. Subsequentemente, na linha 3, para obter essa nota, essa lista é acessada na segunda posição, já que o valor de `self.SECOND_GRADE_INDEX` é 1. Dessa maneira, a nota de interesse é armazenada na variável `second_grade`, cujo valor é inserido de volta na linha referente a redação em questão (variável `example`) dentro da coluna `"second_grade"`, onde todas essas notas estarão armazenadas ao final desse processo.

Ao final desses três passos, como se pode ver na linha 8 do programa 4.13, o resultado das transformações feitas no *dataset* `self.original_dataset` são armazenadas num novo atributo do objeto `self`, chamado `train_dataset`. Por fim, na linha 8, as colunas presentes no atributo `self.columns_to_remove` são removidas do desse novo *dataset*, e então ele é retornado pelo método `preprocess_dataset` dentro de um dicionário de *datasets*, sendo o valor da chave `"train"`, como se vê na linha 10.

4.2.3 Calculador de perplexidade

Tendo em mãos todos os dados necessários ao experimento já prontos para serem usados, precisa-se de algum mecanismo para ler os dados das redações e calcular a sua perplexidade, a fim de possibilitar a validação da hipótese mencionada na seção 4.2.1.

Com essa finalidade, foi implementada a classe `PerplexityCalculator`, cujo código está no arquivo *correction/cluster_of_words/perplexity_calculator.py*. Abaixo, no programa 4.16,

estão os atributos estáticos dessa classe e seus respectivos valores, os quais são apresentados aqui para um melhor entendimento das próximas partes da sua lógica.

Programa 4.16 Atributos estáticos do calculador de perplexidade

```
1  class PerplexityCalculator:
2      text_column = "essay_text"
3      perplexity_column = "perplexity"
4      grade_column = "second_grade"
5      id_column = "essay_id"
```

Além dos atributos estáticos, um objeto dessa classe também possui outros três atributos, cujos valores são definidos no momento da construção do objeto, ou seja, durante o seu método `__init__`, o qual está ilustrado a seguir no programa 4.17.

Programa 4.17 Método construtor do calculador de perplexidade

```
1  def __init__(self, checkpoint: str):
2      self.device = torch.device("cuda") if torch.cuda.is_available() else torch.
           device("cpu")
3      self.model = BertLMHeadModel.from_pretrained(checkpoint, is_decoder=True).
           to(self.device)
4      self.tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

O primeiro deles é o atributo `self.device`, definido na linha 2, o qual pode ser um objeto do tipo `torch.device`, ou seja, a abstração de um dispositivo no qual algum modelo pré-treinado será executado. No caso deste calculador de perplexidade, esse dispositivo pode ser de dois tipos, CPU(obtido com `torch.device("cpu")`) ou TPU/GPU(obtido com `torch.device("cuda")`), da mesma forma que o dispositivo atribuído à rede neural na classe `TopicDeviationFineTuner` (vide 4.1.3).

Já o segundo atributo na linha 3, `self.model`, é um modelo na arquitetura BERT inicializado com o `checkpoint` *neuralmind/bert-base-portuguese-cased*. Diferente do experimento anterior, em que foi utilizado o método `AutoModel.from_pretrained`, o qual retorna um modelo do tipo `BertModel`, aqui foi utilizado o método `BertLMHeadModel.from_pretrained`, o qual retorna um modelo do tipo `BertLMHeadModel`.

A diferença entre esses dois tipos de modelo é que o primeiro é um modelo BERT básico, o qual pode ser treinado para uma ampla gama de tarefas, enquanto que o segundo é um modelo BERT que tem uma camada adicional, a qual serve o propósito de predições relacionadas à tarefa de *language modeling* (Jurafsky e Martin, 2020). Foi feita a escolha por esse segundo tipo, pois ele já realiza a tarefa que precisava-se para esse experimento, sem a necessidade de se treinar um modelo básico do zero para essa tarefa.

Por fim, também é armazenado o atributo `self.tokenizer` na linha 4, o qual será um *tokenizer* específico para o *checkpoint* do modelo instanciado anteriormente.

No *notebook* deste experimento, depois de se obter um objeto do tipo `PerplexityCalculator` chamando o método construtor explicado anteriormente, a fim de validar a hipótese inicial,

Programa 4.18 Método que calcula a perplexidade de todas as redações de um *dataset*

```
1   def calculate_perplexity_for_all_essays(self, df: pd.DataFrame):
2       perplexities = []
3       progress_bar = tqdm(range(len(df.index)))
4
5       for essay in df[self.text_column].tolist():
6           p = self.calculate_text_perplexity(essay)
7           perplexities.append(p)
8           progress_bar.update(1)
9
10      df[self.perplexity_column] = perplexities
11      df = df.sort_values(by=[self.grade_column, self.perplexity_column]).
            reset_index(drop=True)
12      return df
```

chama-se o seu método `calculate_perplexity_for_all_essays` o qual calcula a perplexidade de todas as redações passadas como parâmetro dentro de um `pandas.DataFrame`.

Esse método, apresentado no programa 4.18 acima, inicia, na linha 2, criando uma lista que conterá os valores da perplexidade para cada redação, chamada `perplexities`. Em seguida, na linha 3, também cria uma barra de progresso, `progress_bar` com o auxílio do pacote `tqdm`, o qual irá a renderizar embaixo da célula do *notebook* no qual o método estiver sendo executado.

Depois, na linha 5, itera-se sobre a lista de redações `df[self.text_column].tolist()` , sendo que cada redação é armazenada na variável `essay` a cada iteração. Dentro do laço, faz-se o cálculo da perplexidade para cada redação na linha 6 através do método `calculate_text_perplexity`, cujo resultado é armazenado na variável `p`. Esta é colocada ao final da lista `perplexities` por meio do método `append` presente na linha 7, e então a barra de progresso é atualizada na linha 8.

Ao final da execução desse laço, na linha 10, as perplexidades calculadas são inseridas no *dataframe* `df` passado como parâmetro para o método, fazendo com que uma linha sua contenha uma redação e a sua perplexidade correspondente.

Agora, a fim de entender de fato como cada perplexidade foi calculada, explicar-se-á o método `calculate_text_perplexity` mencionado, cuja implementação está presente no programa 4.19 abaixo.

Ele recebe como parâmetro uma redação `text`, a qual é passada pelo *tokenizer* da classe a fim de transformar as suas palavras em números que as representam utilizando o método `encode` na linha 2. O valor `"pt"` passado para o parâmetro `return_tensors` indica que se almeja receber como saída tensores no formato da biblioteca *PyTorch*, enquanto que o parâmetro `max_length` é o número máximo de *tokens* que serão entregues ao modelo.

Na sequência, essa representação numérica das palavras, `input_ids`, é passada pelo modelo na linha 5, e na linha 6 obtém-se a perda(`outputs.loss`) calculada para ela, a qual é armazenada na variável `log_likelihood`. Por fim, para se alcançar a perplexidade(vide seção 2.2.3) basta elevar o número de Euler, *e*, ao expoente negativo `-log_likelihood`, o que é realizado por meio da função `numpy.exp` na linha 8.

Programa 4.19 Método que calcula a perplexidade de uma redação

```
1   def calculate_text_perplexity(self, text: str):
2       input_ids = self.tokenizer.encode(text, return_tensors="pt", max_length
            =512).to(self.device)
3
4       with torch.no_grad():
5           outputs = self.model(input_ids, labels=input_ids)
6           log_likelihood = outputs.loss.item()
7
8       perplexity = np.exp(-log_likelihood)
9
10      return perplexity
```

4.2.4 Validação da hipótese

Após calcular a perplexidade de todas as redações do *dataset*, foi necessário criar visualizações a respeito desses dados a fim de se compreender se a hipótese levantada era válida. Com esse propósito, foi implementada a classe `PlotterForClusterOfWords`, presente no arquivo *correction/cluster_of_words/plotter.py*, a qual fornece métodos para construir alguns tipos diferentes de visualizações.

O primeiro tipo de visualização criado, através do método `plot_perplexity_for_each_essay`, foi um gráfico de linha, cujo eixo horizontal contém a contagem das redações, e cujo eixo vertical possui o valor numérico da perplexidade de cada redação. Abaixo, na figura 4.1, está um exemplo dessa visualização, o qual foi construído a partir das perplexidades calculadas para o *dataset A*.

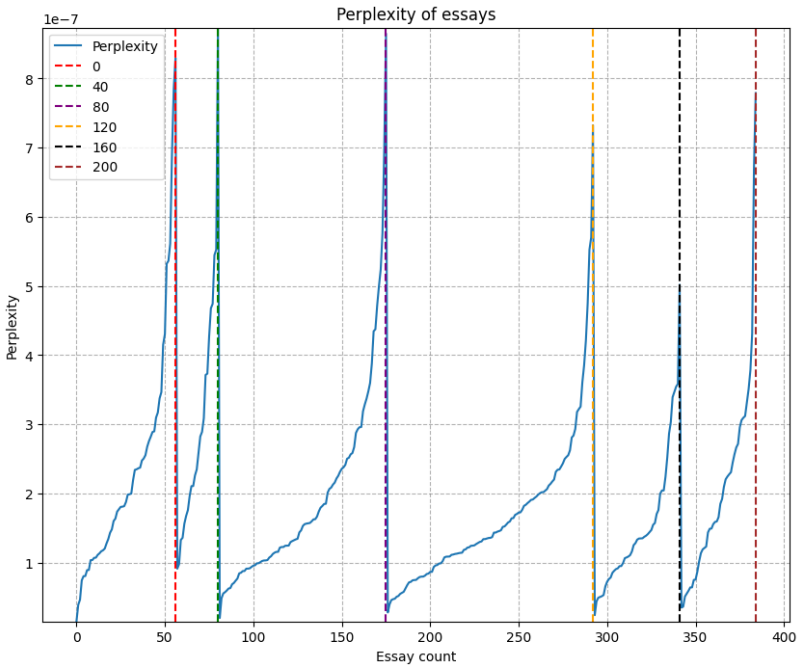


Figura 4.1: *Perplexidade por redação*

Como se pode observar nela, as linhas verticais coloridas separam os grupos de redações

por nota (0, 40, 80, ...). Assim, percebe-se que o valor mínimo de perplexidade para as redações de nota 40(únicas que podem ser classificadas como aglomerados de palavras) é consideravelmente maior do que o mínimo dos demais grupos, o que vai na direção da hipótese levantada.

Além dessa visualização, também foram feitas outras duas, mostrando a média e desvio-padrão da perplexidade de cada um desses mesmos grupos. Tais visualizações seguem abaixo nas figuras 4.2 e 4.3.

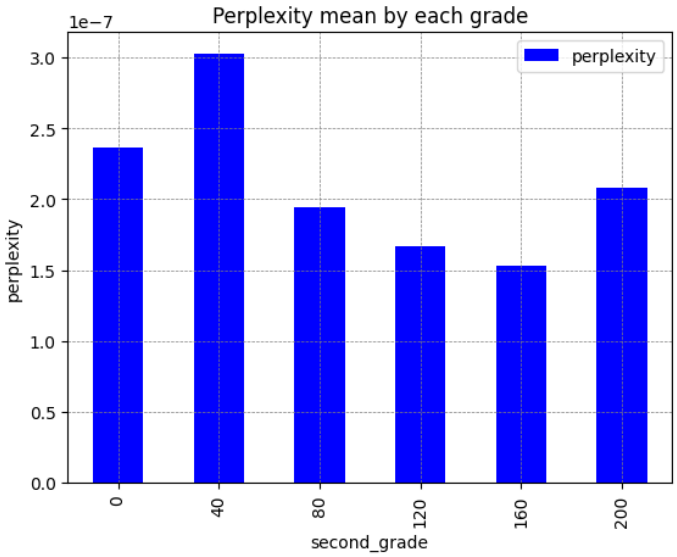


Figura 4.2: *Média da perplexidade das redações por grupo de nota*

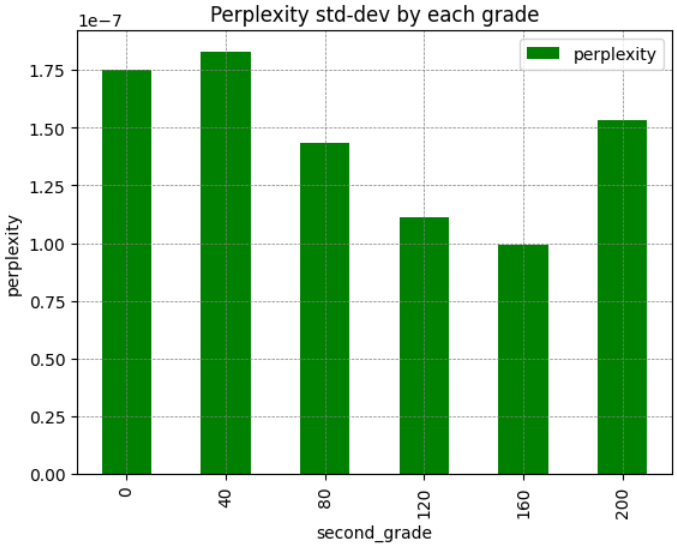


Figura 4.3: *Desvio-padrão da perplexidade das redações por grupo de nota*

Elas foram construídas, respectivamente, pelos métodos `plot_perplexity_mean_for_each_grade` e `plot_perplexity_std_dev_for_each_grade`. Na figura 4.2, é notável a diferença entre a média de perplexidade das redações de nota 40 com relação às demais, sendo

significativamente maior do que elas, o que também leva a acreditar que a hipótese levantada para esse experimento é válida.

Porém, na figura 4.3, nota-se que o desvio-padrão da perplexidade das redações de nota 40 é maior do que as dos outros grupos, o que pode ter uma correlação com o fato de que esse grupo não é composto apenas aglomerados de palavras, mas também por redações que tangenciam o tema e aquelas que possuem traços de outros gêneros textuais (vide seção 2.1), as quais não necessariamente são textos sem um sentido completo nem mau escritos. Dessa forma, suspeita-se que esse desvio-padrão é resultado desses outros dois tipos de redação que compõem esse grupo, os quais podem ser um ruído para esse experimento.

Tendo em vista essas interpretações, decidiu-se seguir com o experimento, considerando essa abordagem uma forma válida para classificar uma redação como um aglomerado de palavras, sendo que a estratégia de classificação será descrita a seguir.

4.2.5 Estratégia de detecção e seus resultados

A ideia da avaliação implementada é que, dado o texto de uma redação, calcula-se a sua perplexidade, e caso essa perplexidade esteja num certo intervalo predeterminado, pode-se afirmar com uma determinada confiança que a nota dessa redação para a competência 2 é 40, e que há uma grande chance de ela ser um aglomerado de palavras.

Para definir qual seria esse intervalo, foram feitos dois testes, cada um utilizando um *dataset* diferente como ponto de partida para o cálculo dos intervalos. Em ambos os testes, para algumas confianças(probabilidades) predeterminadas, calculou-se qual seria o intervalo de perplexidade dentro do qual uma redação seria considerada um aglomerado de palavras com a confiança em questão.

O limite inferior desses intervalos é calculado, porém, como se deseja classificar as redações como aglomerados ou não aglomerados e a hipótese levantada(4.2.1) é a de que quanto maior a perplexidade, maior a probabilidade de a redação ser um aglomerado, não faz sentido calcular um limite superior, pois se a perplexidade de uma dada redação foi maior ou igual ao limite inferior, então ela será classificada como um aglomerado.

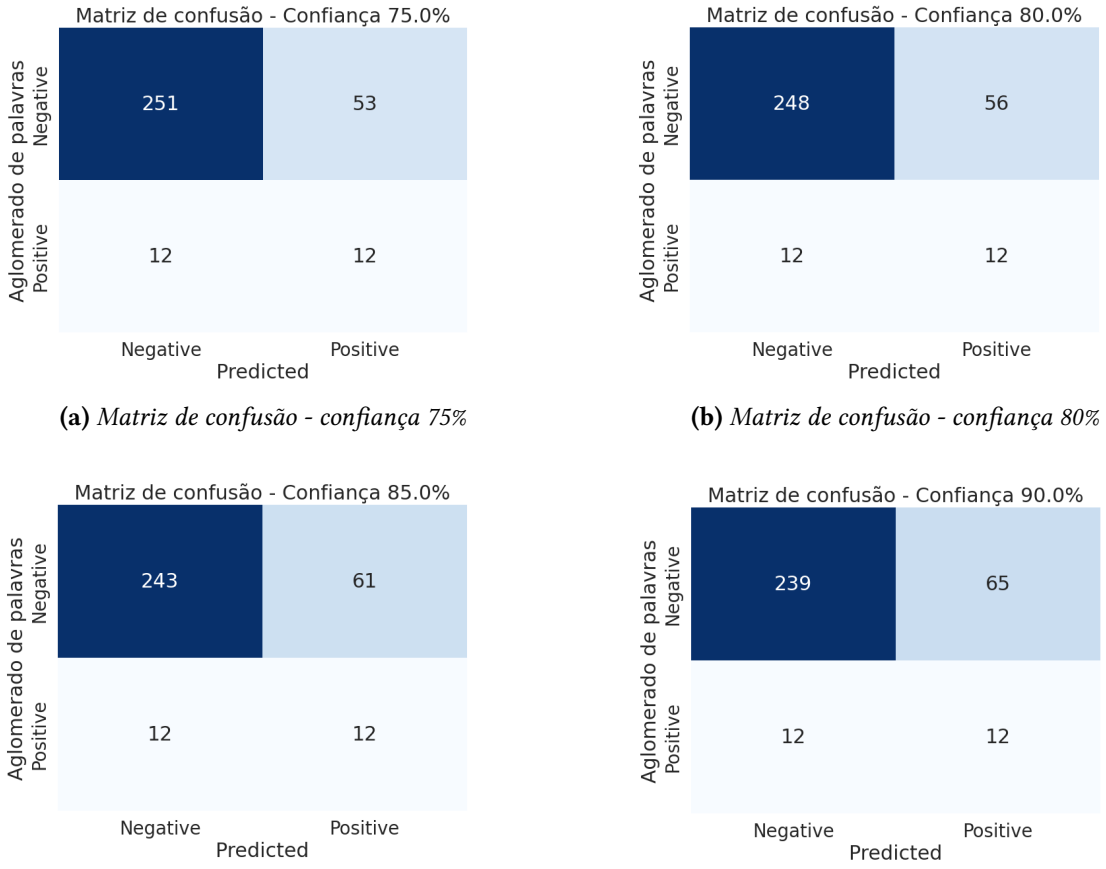
O primeiro teste foi feito utilizando-se como base o *dataset A*. Com o auxílio do método `calculate_confidence_intervals` da classe `PerplexityCalculator`, foram calculados os seguintes intervalos para as suas respectivas confianças:

Confiança	Intervalo
75%	$[0.0000002584, +\infty)$
80%	$[0.0000002532, +\infty)$
85%	$[0.0000002469, +\infty)$
90%	$[0.0000002385, +\infty)$

Tabela 4.2: *Intervalos de confiança para classificação de aglomerados - teste 1*

Tendo esses intervalos em mãos, utilizou-se o método `plot_confusion_matrix_for_confidence_intervals` da classe `PlotterForClusterOfWords` a fim de construir uma matriz de

confusão por intervalo, mostrando como seria o desempenho dessa estratégia para cada um dos intervalos. Abaixo, nas figuras 4.4a, 4.4b, 4.4c e 4.4d, estão as matrizes criadas por esse método:



(a) *Matriz de confusão - confiança 75%*

(b) *Matriz de confusão - confiança 80%*

(c) *Matriz de confusão - confiança 85%*

(d) *Matriz de confusão - confiança 90%*

Figura 4.4: *Matrizes de confusão* - dataset *A* - *teste 1*

Observando essas matrizes, podemos concluir que o melhor intervalo a ser utilizado é o de confiança 75%, pois foi ele teve a melhor acurácia, tanto na classificação de redações que possuem nota 40, quanto para as demais. Para as redações de nota 40, ele atingiu um acerto de 50%. Já para as demais, o acerto foi de 78%.

Dessa forma, a fim de validar se de fato esse seria o melhor intervalo para a aplicação dessa estratégia de avaliação, repetiu-se o processo da construção das matrizes de confusão utilizando os mesmos intervalos, porém com um *dataset* diferente, neste caso, o *dataset B*. Nessa repetição, os resultados que mais chamaram a atenção foram as matrizes dos intervalos de 75% e 90% de confiança, apresentados nas figuras 4.5 e 4.6 a seguir:

Usando a melhor configuração escolhida para o *dataset A*, neste caso, obteve-se uma acurácia de 77% para redações que não são aglomerados, e 50% para redações de nota 40 (possíveis aglomerados), um resultado muito próximo ao do encontrado para o *dataset A*.

Entretanto, neste caso, observando os demais intervalos, nota-se, por exemplo, que o de 90% de confiança traz um aumento de 3% na acurácia da predição dos possíveis

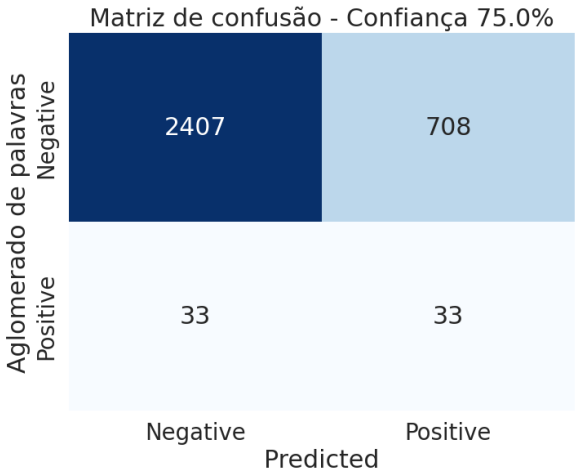


Figura 4.5: *Matriz de confusão - confiança 75%* - dataset *B* - *teste 1*

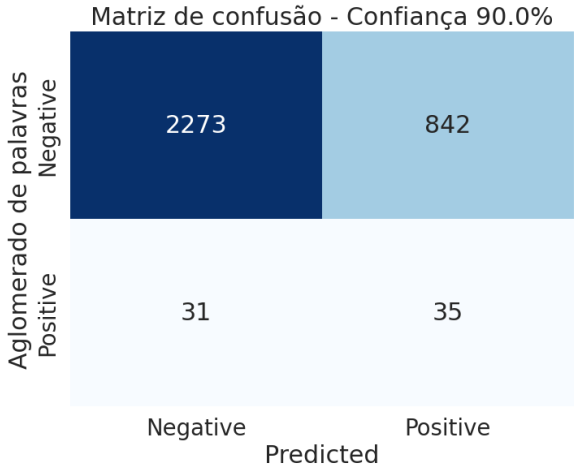


Figura 4.6: *Matriz de confusão - confiança 90%* - dataset *B* - *teste 1*

aglomerados (53%) e uma diminuição de 5% para as redações que não são aglomerados (73%). Portanto, neste primeiro teste, o intervalo escolhido foi o de confiança 75%.

Todavia, a fim de fazer mais uma validação, realizou-se o segundo teste, o qual consistiu no mesmo processo descrito no primeiro, porém invertendo-se a ordem dos *datasets*. Ou seja, os intervalos de confiança foram calculados a partir do *dataset B*, e depois estes intervalos foram utilizados para fazer a avaliação de ambos os *datasets*. Os novos intervalos de confiança calculados se localizam na tabela 4.3 a seguir.

Nesse segundo teste, ao avaliar o *dataset B*, como se pode observar abaixo na figura 4.7, o intervalo que melhor captura as redações que são possíveis aglomerados é o de 90% de confiança. Utilizando ele, obtém-se uma acurácia de 72% para redações que não são aglomerados e 53% para as que são possíveis aglomerados.

Embora a confiança deste intervalo(90%) seja diferente da confiança do intervalo escolhido no primeiro teste(75%), o limite inferior deste intervalo foi muito próximo ao do primeiro, tendo uma diferença de apenas $0,42\%$. Por conta disso, ao realizar a avaliação no

dataset A utilizando esse intervalo, o resultado foi muito semelhante ao do primeiro teste, trazendo um acerto de 50% para as redações de nota 40, e de 78% para as demais.

Confiança	Intervalo
75%	$[0.0000002741, +\infty)$
80%	$[0.0000002645, +\infty)$
85%	$[0.0000002528, +\infty)$
90%	$[0.0000002375, +\infty)$

Tabela 4.3: *Intervalos de confiança para classificação de aglomerados - teste 2*

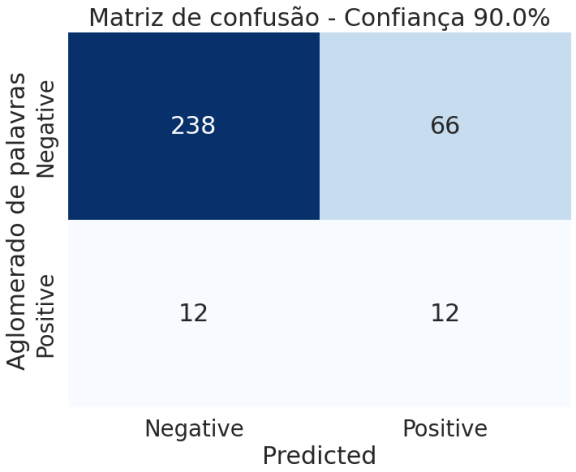


Figura 4.7: *Matriz de confusão - confiança 90%* - dataset *B* - *teste 2*

Como ambos os teste convergiram para um intervalo muito próximo, escolheu-se o intervalo de 90% de confiança da tabela 4.3 para se realizar a detecção de aglomerados de palavras, por ter sido calculado com base em um *dataset* com uma quantidade significativamente maior de dados (*dataset B*).

Capítulo 5

Considerações finais

Este trabalho foi realizado com dois intuitos: um didático, ou seja, para que o autor pudesse aprender conceitos e ferramentas a respeito da área de aprendizado de máquina, e um de pesquisa, isto é, a fim de que se alcançassem avanços na área da avaliação automática de redações do ENEM.

No aspecto didático, acredita-se que foi possível atingir um entendimento considerável de diversos conceitos, técnicas e algoritmos fundamentais voltados ao aprendizado de máquina, mais especificamente, à subárea do PLN, por meio da leitura do livro Jurafsky e Martin, 2020 e também da realização do curso oficial do projeto *Hugging Face* (Face, 2022).

Essas duas fontes de conhecimento foram essenciais para o desenvolvimento do projeto, pois comecei o trabalho sem nenhum conhecimento na área de aprendizado de máquina. Tê-lo terminado conseguindo aplicar os conceitos aprendidos no desenvolvimento das estratégias de avaliação e atingindo bons resultados nas mesmas traz um sentimento de dever cumprido.

Já em relação ao aspecto da pesquisa, pode-se observar que ambos os experimentos aqui desenvolvidos trouxeram avanços significativos, e podem ser refinados em trabalhos futuros.

No experimento da detecção de fuga ao tema, obter uma acurácia de cerca de 82% indica que a técnica de *fine-tuning* aplicada ao modelo pré-treinado é promissora, considerando que ela foi aplicada com um *dataset* poucas centenas de exemplos. Caso seja possível obter *datasets* ainda maiores, é esperado que essa acurácia cresça, e que, dessa forma, seja possível utilizar esse modelo para auxiliar os professores na correção oficial do ENEM, a fim de que consigam facilmente descartar redações que fugiram ao tema, atribuindo-lhes a nota 0.

Para a detecção de aglomerado de palavras, a estratégia empregada requer um grande aprimoramento para atingir um desempenho aceitável em situações reais. Embora ela tenha alcançado uma boa acurácia(78%) para a classificação de redações que não são aglomerados, sua acurácia para a classificação de aglomerados foi 50%, o que significa, neste caso, que ela não é melhor do que um sorteio aleatório de categoria. De qualquer forma, ela também

poderia auxiliar na correção real de redações a fim de descartar a possibilidade de que uma redação é um aglomerado.

Espera-se que este trabalho seja útil para futuros trabalhos com objetivos semelhantes. Algo que ajudaria muito a alcançar resultados melhores em ambas as estratégias apresentadas, seria a obtenção de uma base maior de dados de redações. Dessa forma, sugere-se, para trabalhos futuros, que seja implementada uma ferramenta aberta que permita aos estudantes enviar redações com suas respectivas notas, com a finalidade de acumular dados reais do ENEM e proporcionar um melhor treinamento para modelos semelhantes aos apresentados neste projeto.

Referências

[Albuquerque 2023] Diego Albuquerque. *Li 200 textos por dia e não recebi: Corretores do ENEM denunciam atrasos*. Acessado em 4 de maio de 2023. 2023. url: https://www.bol.uol.com.br/noticias/2023/05/04/li-200-textos-por-dia-e-nao-recebi-corretores-do-enem-denunciam-atrasos.htm (citado na pg. 2).

[Devlin *et al.* 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee e Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL] (citado na pg. 9).

[Educação 2012] Ministério da Educação. *Correção*. Website. http://portal.mec.gov.br/component/tags/tag/correcao. 2012 (citado nas pgs. 1, 5).

[Face 2022] Hugging Face. *The Hugging Face Course, 2022*. https://huggingface.co/course. Online; acessado em 20 de outubro de 2023. 2022 (citado nas pgs. 28, 45).

[Foundation 2024] Python Software Foundation. *Python Documentation*. Accessado em: 24 de janeiro de 2024. 2024. url: https://docs.python.org/3.12/reference/index.html (citado na pg. 12).

[INEP 2020] INEP. *Competência 2 do ENEM - Módulo 4*. 2020. url: https://download.inep.gov.br/educacao_basica/enem/downloads/2020/Competencia_2.pdf (citado na pg. 6).

[Jurafsky e Martin 2020] Daniel Jurafsky e James H. Martin. *Speech and Language Processing*. Stanford University, 2020 (citado nas pgs. 7, 9–11, 26, 27, 31, 37, 45).

[Lipton *et al.* 2015] Zachary C. Lipton, John Berkowitz e Charles Elkan. *A Critical Review of Recurrent Neural Networks for Sequence Learning*. 2015. arXiv: 1506.00019 [cs.LG] (citado na pg. 9).

[Loshchilov e Hutter 2019] Ilya Loshchilov e Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: 1711.05101 [cs.LG] (citado na pg. 29).

[Ruder *et al.* 2019] Sebastian Ruder, Matthew E. Peters, Swabha Swayamdipta e Thomas Wolf. "Transfer learning in natural language processing". Em: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*. Ed. por Anoop Sarkar e Michael Strube. Minneapolis, Minnesota: Association for Computational Linguistics, jun. de 2019, pp. 15–18. doi: 10.18653/v1/N19-5004. url: https://aclanthology.org/N19-5004 (citado na pg. 9).

[Silveira *et al.* 2024] Igor Cataneo Silveira, André Barbosa e Denis Deratani Mauá. "A new benchmark for automatic essay scoring in portuguese". Em: *16a Conferência Internacional em Processamento Computacional de Português (PROPOR 2024)*. Aceito. Mar. de 2024 (citado na pg. 15).

[Souza *et al.* 2020] Fábio Souza, Rodrigo Nogueira e Roberto Lotufo. "Bertimbau: pretrained bert models for brazilian portuguese". Em: out. de 2020, pp. 403–417. isbn: 978-3-030-61376-1. doi: 10.1007/978-3-030-61377-8_28 (citado nas pgs. 9, 26).

[Vaswani *et al.* 2017] Ashish Vaswani *et al. Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL] (citado nas pgs. 7, 8).

[Weisstein 2024] Eric W. Weisstein. *Harmonic Mean*. Acessado em: Janeiro 24, 2024. 2024. url: https://mathworld.wolfram.com/HarmonicMean.html (citado na pg. 10).