Análise de Sentimentos no Canto 1 da Ilíada

Rafael Oleques Nunes [1] **, Vitória Colonetti Benedet** [1] **, Rafael Fernandes Borges** [1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

1. Definição do trabalho

Nosso trabalho tem como objetivo desenvolver um modelo baseado em aprendizado de máquina para prever os sentimentos (positivo, negativo, neutro ou narrador) expressos no primeiro canto do épico grego *Ilíada* a partir da tradução dos versos em grego moderno. Para tanto, utilizamos o conjunto de dados de 611 versos com as suas respectivas classificações retiradas do artigo *Sentiment Analysis of Homeric Text: The 1st Book of Iliad* [Pavlopoulos et al. 2022] em que os dados estão disponíveis no Github[1].

2. Dataset

Nesta seção iremos discutir sobre o domínio escolhido e as característica do dataset utilizado. Será abordado sobre exemplos do texto do primeiro canto da Ilíada na Subseção 2.1 e características do dataset, como valores faltantes, escolha e distribuição das classes na Subseção 2.2.

A título de melhor entendimento, todos os versos citados neste trabalho são retirados da tradução de Frederico Lourenço publicada pela editora Penguin [Homero 2013].

2.1. Domínio dos dados

A Ilíada é um épico grego atribuído ao poeta Homero. O poema é ambientado durante o nono ano da Guerra de Tróia e possui como tema central a narração da fúria de Aquiles e das suas consequência, como é indicado no primeiro verso, no qual o poeta nos dá o argumento de sua obra e pede a inspiração da musa: *Canta, ó deusa, a cólera de Aquiles, o Pelida.*

O poema possui 24 cantos em que cada um é dividido em versos. Neste trabalho o foco é no primeiro canto que possui 611 versos. Dentro dos cantos ocorrem tanto cenas de diálogos diretos entre os personagens quanto narrativas com falas do narrador, paráfrase de falas dos personagens, descrição de ambiente e ação dos personagens.

Um exemplo de diálogo pode ser visto nos versos de 17 à 21 do canto I em que temos a fala do sacerdote Crises pedindo o resgate de sua filha:

> "Ó Atridas e vós, demais Aqueus de belas cnêmides!
> Que vos concedam os deuses, que o Olimpo detêm,
> saquear a cidade de Príamo e regressar bem a vossas casas!
> Mas libertai a minha filha amada e recebei o resgate,
> por respeito para com o filho de Zeus, Apolo que acerta ao longe."

[1]https://github.com/ipavlopoulos/sentiment_in_homeric_text

Ilíada, I, 17-21

Um exemplo de narração é vista nos versos de 22 à 25 do canto I em que temos a descrição da reação dos gregos ao pedido de Crises:

> Então todos os outros Aqueus aprovaram estas palavras:
> que se venerasse o sacerdote e se recebesse o glorioso resgate.
> Mas tal não agradou ao coração do Atrida Agamêmnon;
> e asperamente o mandou embora, com palavras desabridas

Ilíada, I, 22-25

2.2. Características dos dados

O dataset possui 611 instâncias em que cada uma corresponde a um dos versos do primeiro canto da Ilíada. Os dados foram coletados pelos autores do artigo usado como base, os quais realizaram questionários com nativos de grego. O texto armazenado está em grego moderno, tal escolha é justificada, pois o grego moderno tem uma proximidade maior do grego antigo em relação às outras línguas modernas, trazendo, assim, uma maior proximidade com o texto original.

Originalmente o dataset possui seis colunas: *greek text*, *neutral*, *positive*, *negative*, *narrator* e *y*, sendo, respectivamente, o texto do verso em grego moderno, a porcentagem de pessoas que classificaram o verso como neutro, porcentagem de pessoas que classificaram como positivo, porcentagem de pessoas que classificaram como fala do narrador e um vetor com estes mesmos valores. Adicionamos no dataset mais uma coluna *class* com o nome da classe que será utilizada como atributo alvo.

Para a classificação utilizamos a coluna *greek text* para extrair os valores de entrada para os modelos. A coluna *class* como atributo alvo para obter a *string* com os possíveis sentimentos do verso (*neutral*, *positive*, *negative* e *narrator*), sendo esta categórica.

2.3. Atributo alvo

Originalmente não temos a informação direta da classe de um verso no dataset, uma vez que nem sempre se tem um consenso forte sobre qual é a classe correta de um verso, como podemos ver na Tabela 1 nos versos número 44 e 46 em que se tem duas classes com 50% de escolha. Tendo em vista esta problemática, a abordagem de utilizar as probabilidades no processo de treinamento foi escolhida pelos autores para que se pudesse emular de forma mais precisa o resultado obtidos pelo grupo de anotadores.

Por simplificação, neste trabalho está sendo utilizada a classe com a maior porcentagem como classe alvo. Ainda, assim, foram encontrados 69 casos em que está ambígua qual deve ser a classe alvo, uma vez que mais de uma classe possui a maior porcentagem, para resolver este problema foi utilizada a primeira classe que possui o maior valor. Ainda que esta seja uma abordagem simples, podemos observar na Figura **??** que conseguimos obter uma boa coesão de sentimentos entre versos consecutivos, uma vez que esta ambiguidade tende a ocorrer em versos seguidos, como vemos no exemplo da Tabela 1.

Após as operações citadas, o dataset possui 266 instâncias da classe *positive*, 232 instâncias da classe *negative*, 74 instâncias da classe *neutral* e 39 instâncias da classe *narrator*, como podemos ver na distribuição apresentada na Figura 3. Ao realizar está

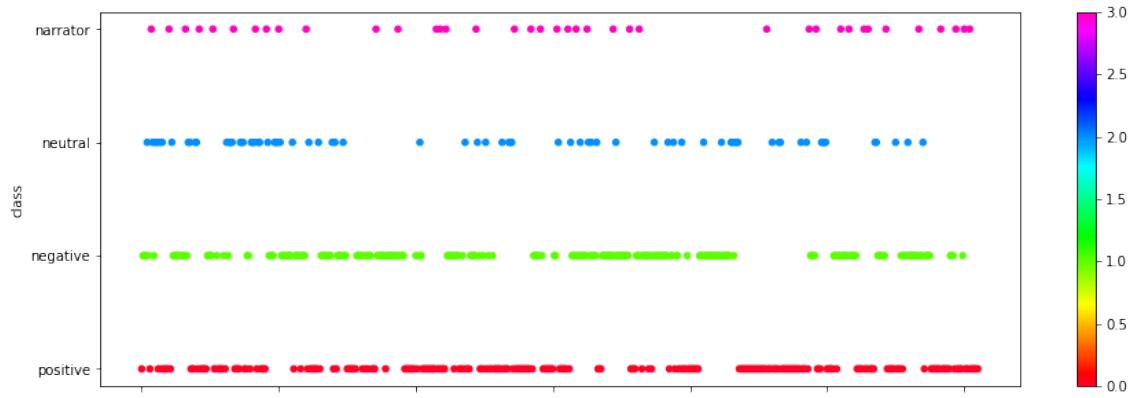

Figura 1. Scatterplot mostrando os sentimentos expressos ao longo dos versos. No eixo x temos o número dos versos em ordem crescente e no eixo y temos o sentimento que está sendo expresso, as cores servem para reforçar visualmente o valor do sentimento do verso.

análise da distribuição do dataset, foi constatado que se tinha duas classes minoritárias – *neutral* e *narrator* –, tendo em vista esta problemática decidimos utilizar diferentes abordagens para lidar com este problema, como é abordado na Subseção 4.3.

3. Setup

Para o desenvolvimento deste trabalho utilizamos a linguagem de programação Python na versão 3.9.16. Utilizamos a biblioteca Pandas para processarmos o CSV, a biblioteca Numpy para representação vetorial e para manipulação dos vetores, também utilizamos a biblioteca Random para realizar a operação de *shuffle* sobre eles. Para melhor entender a distribuição dos dados e visualizar de forma gráfica os dados utilizamos as bibliotecas Matplotlib e Seaborn.

Os modelos de Aprendizado de Máquina foram obtidos por meio da biblioteca scikit-learn com suas implementações padrão. A conversão dos textos para dados numéricos foi feita por meio da implementação da matriz TF-IDF (*Term Frequency – Inverse Document Frequency*) e a conversão dos dados catégóricos por meio de *One Hot Enconding*, ambas também oferecidas pela scikit-learn. Utilizamos também uma abordagem que leva em conta o contexto das palavras por meio de embeddings gerados pelo modelo SBERT (*Sentence-BERT*) [Reimers and Gurevych 2019] disponíveis na biblioteca sentence-transformers.

4. Metodologia

Nesta seção iremos descrever a metodologia utilizada neste trabalho, tratando sobre a utilização de funções prontas, as funções que nós implementamos e a combinação destas para gerar as predições. Na Subseção 4.1 tratamos sobre as abordagens de representação vetorial dos dados de entrada, na Subseção 4.2 tratamos sobre abordagens de pré processamento de texto antes de gerar as representações vetoriais, na Subseção 4.3 tratamos sobre manipulações que fizemos nas classes alvos e enfrentamento do problema de duas classes mitoritárias, na Subseção 4.4 tratamos sobre os modelos de aprendizado de máquina utilizados, na Subseção 4.5 tratamos sobre a nossa implementação da validação cruzada,

Nro.	Grego	Português	Neutro	Positivo	Negativo	Narrador
43	Είπε, και την ευκή του επάκυσεν Απόλλωνας Φίβς,	Assim disse, orando; e ouviu-o Febo Apolo.	0.0	0.0	0.0	1.0
44	κι απ την κρφή του λύμπου εχύθηκε θυμό γεμάτς, κι είχε	Desceu do Olimpo, com o coração agitado de ira.	0.0	0.5	0.5	0.0
45	δξάρι και κλειστό στις πλάτες του περάσει σαϊτλόγ'	Nos ombros trazia o arco e a aljava duplamente coberta;	0.125	0.5	0.25	0.125
46	κι αντιβρντύσαν ι σαγίτες του στις πλάτες, μανιασμένς	aos ombros do deus irado as setas chocalhavam à medida que avançava.	0.0	0.5	0.5	0.0
47	καθώς τραβύσε· και κατέβαινε σαν τη νυχτιά τη μαύρη.	E chegou como chega a noite.	0.125	0.5	0.25	0.125

Tabela 1. Porcentagem dos sentimentos anotados nos versos em que Crises termina sua prece a Apolo e o deus desce do Olimpo para vingar seu sacerdote e espalhar uma praga no exercito grego. Nota-se que o verso 44 e 46 possuem ambiguidade em relação ao sentimento expresso.

na Subseção 4.6 tratamos sobre a implementação das métricas e, por fim, na Subseção 4.7 tratamos sobre a integração destas funções em nosso *pipeline*.

4.1. Representações de texto como vetores

Modelos como Random Forest, Decision Tree e KNN são implementados de forma que recebem valores numéricos como parâmetros, entretanto, neste trabalho estamos utilizando textos como atributos. Para resolver este problema buscamos técnicas de pré processamento que convertem textos para vetores numéricos. Dentre as técnicas encontradas escolhemos utilizar duas técnicas de representação vetorial: TF-IDF e SBERT.

A matriz TF-IDF possui em suas linhas o índice de um documentos e nas suas colunas o índice de um *token*, sendo um documento um texto qualquer e um token uma palavra qualquer presente em pelo menos em um dos documentos. Diferente da abordagem *Bag of Words* que armazena se uma palavra ocorre ou não nos textos, em um vetor TF-IDF temos a frequência em que ela ocorre. Entretanto, não temos o valor bruto da frequência – isso ocorre em vetores TF (*Term-Frequency*) –, mas o valor ponderado da palavra em relação a um documento em questão e aos documentos em que ela aparece. Na prática, temos que palavras que aparecem muito em vários documentos são penalizadas

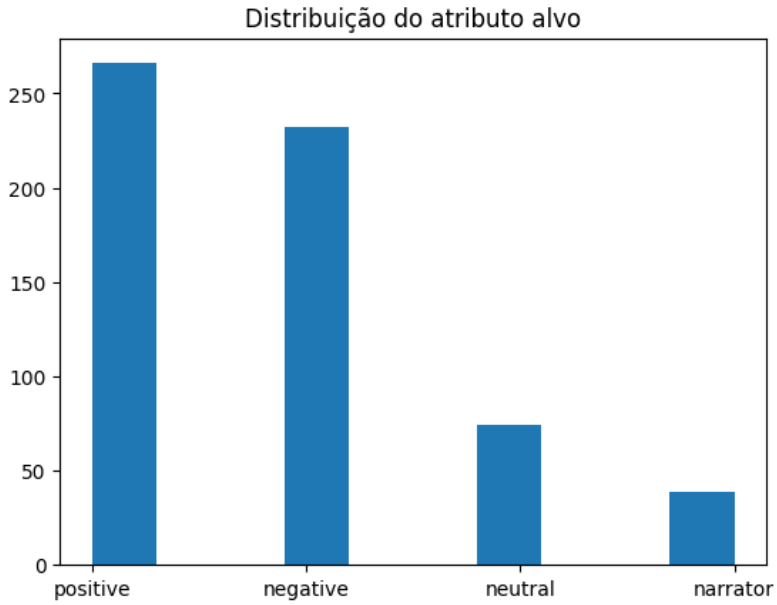

Figura 2. Distribuição do atributo alvo em nosso dataset. Podemos observar que temos como classes predominantes positive e negative, enquanto as classes neutral e narrator são minoritárias.

com um valor baixo por não serem significativas, enquanto palavras que aparecem com maior frequência em um documento, mas pouco no todo, recebem valores maiores por, provavelmente, serem significativas para representar aquela sentença. Por padrão matrizes TF-IDF já possuem seus valores ponderados devido ao cálculo feito para chegar nela, dessa forma, não realizamos normalização de dados de forma explícita nesta abordagem.

Os embeddings gerados pelo SBERT possuem 512 posições, sendo, respectivamente, a representação de cada *token* da sentença que está sendo processada por ele, caso se tenha menos que 512, a sentença é preenchida com *tokens* auxiliares vazios. Diferente do TF-IDF que possui foco na distribuição ao longo de documentos para gerar sua representação vetorial, no SBERT os vetores são gerados lidando também com a semântica e o contexto da sentença, além de também lidar com polissemia, ou seja, representações diferentes de uma mesma palavra e, também, com palavras que possuem significados diferentes em diferentes contextos, como o banco da praça que se refere a um assento e banco de dados que se refere ao armazenamento de dados. O SBERT também tende a não necessitar normalização explícita, então também não aplicamos nestes vetores.

4.2. Pré processamento dos versos

Dentro da área de PLN (Processamento da Linguagem Natural) há diversas técnicas de pré processamento de textos visando com que as representações vetoriais sejam as mais expressivas possíveis. Diferentes formas de representações podem exigir pré processamentos diferentes, desde aquelas que podem se beneficiar de maiores operações sobre os dados, como é o caso de matrizes TF-IDF, até aquelas que aproveitam melhor o texto com o menor número de operações, como representações geradas pelo SBERT.

Com isto, neste trabalho fizemos experimentos aplicando operações de lematização e remoção de *stop words* na matriz TF-IDF e a utilização do texto completo para a matriz

TF-IDF e para os embeddings gerados pelo SBERT. Abaixo, são apresentados maiores detalhes sobre a escolha das operações para cada uma das representações.

Lematização, operação de reduzir um verbo para seu infinitivo, enquanto adjetivos e substantivos são reduzidos para o seu masculino e para o singular. O objetivo da utilização desta operação foi reduzir diferentes representações da mesma palavra na matriz TF-IDF, o que permite com que uma mesma palavra possa ser reconhecida como sendo a mesma, independentemente se está no plural, singular, em outro tempo verbal etc. nas suas diferenças ocorrências nos versos. Além disso, excluindo estas outras representações temos como consequência o ganho na diminuição da dimensionalidade da matriz TF-IDF.

Remoção de stop words, operação em que são removidas de um texto palavras que não tendem a carregar um significado relevante para o texto, como "a", "o", "para", "com", "de"etc. A lista de *stop words* em grego foi extraída da biblioteca Spacy, não houve algum tipo de enriquecimento manual desta lista devido ao desconhecimento da língua grega de nossa parte. Ainda que matrizes TF-IDF já lidem com estas palavras de alta frequência na maioria dos textos, penalizando com valores baixos, elas continuam como colunas na matriz, o que contribui para que se tenha uma maior dimensionalidade. Desta forma, a remoção das *stop words* tem como objetivo a diminuição da dimensionalidade da matriz TF-IDF. Nota-se que em nossos testes sempre utilizamos a lematização junto com esta operação, justamente por esta propriedade de penalização e a possibilidade de reduzir a dimensionalidade.

Utilização do texto completo sem a aplicação nenhum tipo de pré processamento prévio. Neste cenário utilizamos os textos completos para a geração das representações. É importante destacar que representações geradas pelo SBERT lidam melhor com o texto com o menor número de alterações devido as suas propriedades de identificação de contextos, por isso não aplicamos as operações descritas acima quando utilizamos ele.

Além destas operações, consideramos outras abordagens, como *stemming* (conversão de uma palavra para sua raiz) e a utilização dos fonemas das palavras em grego em nosso alfabeto, mas por limitação de tempo não aplicamos em nossa avaliação final.

4.3. Pré processamento da classe alvo

A classe alvo em nosso dataset está representada por meio de um valor categórico: *neutral*, *positive*, *negative* ou *narrator*. Entretanto, isto se torna problemático, tendo em vista que os algoritmos utilizados possuem como saída valores numéricos, não categóricos. A solução encontrada foi utilizar *One Hot Enconding* para gerar representações vetoriais de nossos dados. Inicialmente cogitamos converter as classes para valores entre [-1, 2], mas descartamos esta possibilidade ao perceber que estaríamos inserindo uma noção de ordem que não existe nos dados originais.

Além do problema anteriormente mencionado, encontrou-se mais uma questão, sobre como lidar com duas classes minoritárias em nossa classificação. Com o objetivo de explorar diferentes soluções, testamos três estratégias: utilizar as classes sem nenhuma modificação, remover a classe *narrator* – por possuir o menor número de instâncias – e juntar as duas classes minoritárias em uma nova classe *others*. As três estratégias foram utilizadas no processo de treinamento e seus resultados podem ser conferidos na Seção 5.

4.4. Modelos

Os modelos escolhidos foram árvore de decisão, K-ésimo vizinho mais próximo (KNN) e floresta aleatória. Não foram realizadas otimizações dos hiperparâmetros, sendo empregado os padrões presentes na biblioteca scikit-learn.

4.4.1. Árvore de Decisão

Esse classificador possui como entrada principal argumentos como profundidade máxima, peso da classe e estado aleatório. Foi utilizado os valores padrões:

- profundidade máxima é 5;
- peso da classe ('balanced')
- estado aleatório é 42

4.4.2. KNN

Os principais argumentos que tem como entrada são o número de vizinhos, os pesos, o algoritmo para encontrar os vizinhos mais próximos, a métrica da distância entre vizinhos, o número de trabalhos em paralelo. Como não foi passado parâmetos, foi usado os hiperparâmetros padrão, são eles:

- número de vizinhos é 5;
- peso uniforme ('uniform')
- o algoritmo automático ('auto')
- a métrica é minkowski
- o número de trabalhos em paralelos é 1

4.4.3. Floresta Aleatória

Os principais argumentos que a função aceita como entrada são o número de árvores, a função para medir a qualidade de uma divisão, restrição de profundidade máxima, número mínimo para dividir um nodo, número mínimo de amostra nas folhas, entre outras. Novamente não foi passado nenhum parâmetro, usando assim os valores padrão, sendo os principais:

- o número de árvores é 100
- a função para medir a qualidade de uma divisão é a gini
- sem restrição de profundidade máximo
- número mínimo para dividir um nodo é 2
- número mínimo de amostra nas folhas é 1

4.5. K-fold cross validation

A validação cruzada (*cross validation*) é uma estratégia para avaliar a capacidade de generalização de um modelo. Sua proposta consiste em dividir o conjunto original em k partições (*folds*) disjuntas e de tamanhos aproximadamente iguais. Com essas instâncias, é realizado treinamento e teste de forma iterativa, de maneira que em cada

uma das k repetições, um *folds* diferente é utilizado para teste, e o k-1 *folds* restantes para o treinamento. O desempenho final, então, é dado pela média e desvio padrão dos k conjuntos de testes avaliados.

Foi realizada uma implementação própria da validação cruzada de forma estratificada, a qual foi dividida em três etapas: estratificação dos conjuntos, divisão dos k folds e iteração de treinamento. O pseudo código da divisão dos k folds pode ser visto no Algoritmo 1 e a aplicação da validação cruzada pode ser vista no Algoritmo 2.

Algorithm 1 K-fold division

1: Divide os dados em k folds estratificados
2: **for** $indexFoldTest = 0, 1, \ldots, k$ **do**
3: Faz uma lista de treino com os indexes que estão nos folds que não possuem o índex $indexFoldTest$
4: Faz uma tupla com os índexes treino e indexes do fold de teste e salva na lista $finalFolds$
5: **end for**
6: Retorna a lista $finalFolds$ com uma lista de tuplas que possuem os índexes dos dados que estarão nos folds de treino e folds de teste

Algorithm 2 K-fold Cross Validation

1: Inicializa uma lista para cada métrica
2: Obtém os *folds* através do k-fold division
3: **for** *foldTreino*,*foldTeste* em *folds* **do**
4: Obtém X e y (classe) de treino através do *foldTreino*
5: Obtém X e y (classe) de teste através do *foldTeste*
6: Utiliza X e y de treino para aprender
7: Utiliza X de teste para predizer classe
8: Calcula métricas através da classe predita e da classe verdadeira
9: **end for**
10: Retorna uma lista de cada uma das métricas com os seus valores nos k folds

A estratificação é uma técnica de amostragem que permite que as amostras tenham a mesma proporção de características da população. Na validação cruzada, isso permite que as instâncias utilizadas para treino e teste possuam a mesma proporção de classes do dataset original.

A fim de obter essa proporção, a implementação própria da estratificação, transforma os dados catégóricos que estão em *One Hot Enconding* em *strings* para serem utilizados como chaves de um dicionário. Esse dicionário, portanto, possui como chave a classe e como valor um vetor com os índices do dataset correspondentes. Após todo o conteúdo ser devidamente segmentado, é realizado um *shuffle* para evitar que seja estratificado sempre da mesma forma, e então cada lista de classe é dividida em k partes. Por fim, cada partição da classe é unida com a partição correspondente das outras classes, gerando k amostras com tamanho e proporção aproximadamente iguais, como podemos ver no exemplo da Figura 3 com gráficos de barras representando a porcentagem das classes em cada uma das k partições.

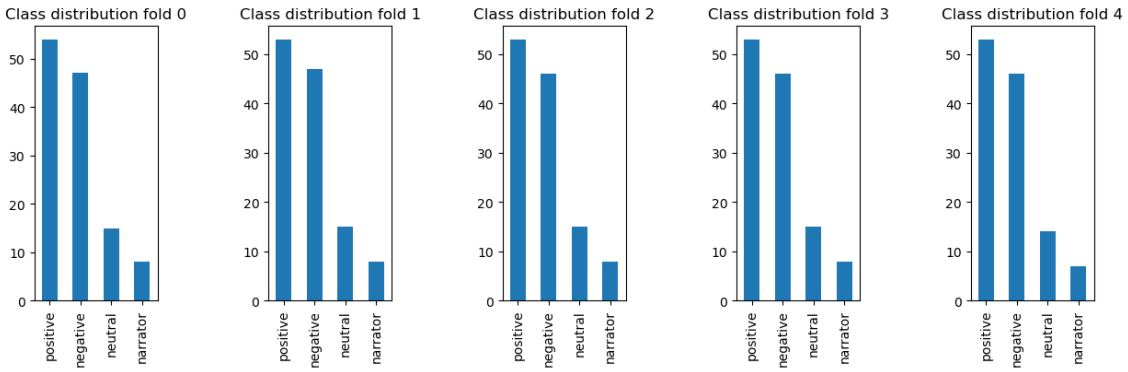

Figura 3. Distribuição dos atributos para k igual a 5. Podemos observar que a distribuição das classes é preservada em relacão ao dataset.

4.6. Matriz de confusão

Usada para calcular as métricas e obter informações importes do treinamento. Por nossos dados se tratarem de multiclasses, no algoritmo começamos construindo uma matriz com as dimensões referente ao número de classes com uma adição de coluna para quando o modelo não conseguir classificar, sendo, por exemplo, 4x5 quando utilizamos todas as nossas classes. Em seguida é testado cada valor original com o resultado do treinamento, isso tendo 3 opções:

- O resultado é igual
- O resultado é diferente
- Não foi possível classificar

Para os dois primeiros itens é incrementado em uma unidade a célula correspondente ao caso. A posição da classe original determina as linhas, enquanto que a posição da classe utilizada no treino determina as colunas. Somente as células com índice de linha e coluna iguais são valores verdadeiro positivo.

Já quando não foi possível classificar, é adicionado uma unidade em uma correspondente a linha na última coluna possível, assim representando um falso positivo, porém não influenciando nos cálculos que utilizam valores das colunas.

4.6.1. Acurácia

Acurácia leva em conta toda a matriz, e, por causa disto, a função simplesmente calcula os valores dos verdadeiros positivos e divide pelo valor total da matriz.

4.6.2. Precisão

Optamos por calcular a precisão macro. Para isso é calculada individualmente a precisão de cada classe dividindo-se o valor de verdadeiros positivos pelo valor total da coluna. Após, é tirada uma média simples do valor dessas precisões, resultando na precisão macro.

4.6.3. Recall

Abordamos o recall de uma forma semelhante a da precisão, calculamos individualmente o valor do recall para cara classe dividindo o valor de verdadeiros positivos pelo número total da linha. No fim tiramos uma média simples dos valores individuais de recall, encontrando o recall macro.

4.6.4. F1-score

Após ter a função da precisão e do recall é possível calcular facilmente o F1-score. É duas vezes o valor da multiplicação do recall pela precisão dividido pela soma do recall e da precisão.

4.7. Pipeline e integração

A princípio o processo para a construção foi feito utilizando as bibliotecas prontas e aos poucos foi sendo introduzido nossos algoritmos. Assim, focamos em criar o pipeline para em seguida integrar nossos algoritmos. Para facilitar os processos de testes com os difrentes cenários, criamos uma classe *Trainer* que consegue simular nossos cenários dependendo dos parâmetros que passamos a ela, tal classe está presente nos arquivos disponibilizados junto do relatório e sua execução pode ser vista no notebook *implementacoes*.

A primeira etapa aplicada é a de pré processamento, sendo divido no pré processamento das classes e no pré processamento do texto. Caso se deseje fazer alterações nas classes é possível utilizar um método para remover uma classe ou para juntar duas em uma nova classe *others*. Em relação ao texto, é possível utilizar as funcionalidades de lematização das frases para logo em seguida remover os stop words, gerar representações do SBERT ou não aplicar nenhum tipo de transformação no texto, dependendo dos parâmetros passados. Após isto, o método que realiza o pré processamento do texto também aplica a função de *one hot enconding* nos valores alvos.

Em seguida, através da função implementada de validação cruzada, é obtido os folds e sua divisão de treino e teste, que por sua vez fornecem os valores X e y de treino e teste. Caso se deseje utilizar a matriz TF-IDF, os textos são convertidos para a mesma por meio da estrutura de *pipeline* do sklearn, senão usa-se os embeddings do SBERT, já pré computados na fase de pré processamento. Assim, os valores numéricos gerados e as classes alvos são passados para o modelo escolhido e realizado o processo de treinamento. Os valores que estão no *fold* de teste são utilizados para realizar a predição do modelo e, então, os resultados são passados como entrada para as funções que montam a matriz de confusão para, em fim, calcular as métricas implementadas e salvá-las em uma lista para cada métrica. Ao fim, as listas de cada uma das métricas são retornadas em forma de dicionário, sendo o nome da métrica utilizada como chave e a lista de k resultados como valor.

Ainda usamos as bibliotecas do Pandas e Numpy em nossas funções, a biblioteca spacy para podermos fazer a lematização das frases e as funções OneHotEncoder, TfidfVectorizer e Pipeline do sklearn.

Código	Cenário
C0	Decision Tree utilizando TF-IDF sem lematização
C1	Decision Tree utilizando TF-IDF com lematização
C2	Decision Tree utilizando SBERT sem lematização
C3	KNN utilizando TF-IDF sem lematização
C4	KNN utilizando TF-IDF com lematização
C5	KNN utilizando SBERT sem lematização
C6	Random Forest utilizando TF-IDF sem lematização
C7	Random Forest utilizando TF-IDF com lematização
C8	Random Forest utilizando SBERT sem lematização

Tabela 2. Códigos dos cenários utilizados.

Para gerar os gráficos para comparações, utilizamos a função *compare_boxplot*, implementada pelo grupo, que gera um *plot* para cada uma das classes em um dos cenários utilizados. Todos os gráficos gerados podem ser consultados no notebook *implementacoes*.

5. Resultados e discussões

Como descrito nas Subseções 4.1 e 4.2, utilizamos diferentes cenários para processar os nossos dados, os quais estão listados na Tabela 2 e, como apresentado na tabela, possuem um código para que seja mais fácil fazer referência a eles nas tabelas dos resultados. Devido ao pequeno número de dados, não otimização de hiperparâmetros e a alta dimensionalidade dos dados, não obtivemos resultados satisfatórios em nossos experimentos.

Todos os valores de acurácia médios obtidos foram menores do que 50%, como pode ser visto na Tabela 3. O melhor resultado foi encontrado no caso em que foi aplicado o modelo de *decision Tree* utilizando TF-IDF com lematização (classe C1) e fazendo a junção da classe *neutral* e *narrator*, no qual obtivemos a a acurácia média de 45.66% e desvio padrão de 2.14. Podemos ver este comportamento na Figura 4.

Ademais, a Figura 4 também ilustra o comportamento geral que ocorreu nos cenários que utilizamos *decision tree* com os valores variando em uma menor escala, intercallando entre os casos da remoção da classe *narrator* e da junção desta classe com *netural*.

Métrica e cenário	Sem modificação	Sem narrador	Junção de narrador e neutro
acurácia (C0)	42.23 ± 5.48	39.34 ± 2.62	42.25 ± 3.55
acurácia (C1)	41.74 ± 1.71	44.56 ± 2.73	45.66 ± 2.14
acurácia (C2)	39.1 ± 5.53	44.39 ± 3.25	42.73 ± 5.06
acurácia (C3)	37.62 ± 2.07	41.41 ± 2.96	39.42 ± 3.97
acurácia (C4)	31.07 ± 4.77	37.93 ± 2.42	33.87 ± 3.62
acurácia (C5)	35.87 ± 4.93	41.1 ± 5.92	39.15 ± 5.87
acurácia (C6)	27.49 ± 1.33	31.82 ± 1.38	27.49 ± 2.71
acurácia (C7)	37.47 ± 2.18	43.0 ± 1.87	39.11 ± 2.56
acurácia (C8)	22.43 ± 2.01	31.46 ± 4.83	23.4 ± 3.53

Tabela 3. Acurácia média e desvio padrão de cada um dos cenários utilizados com a variação de classes alvo.

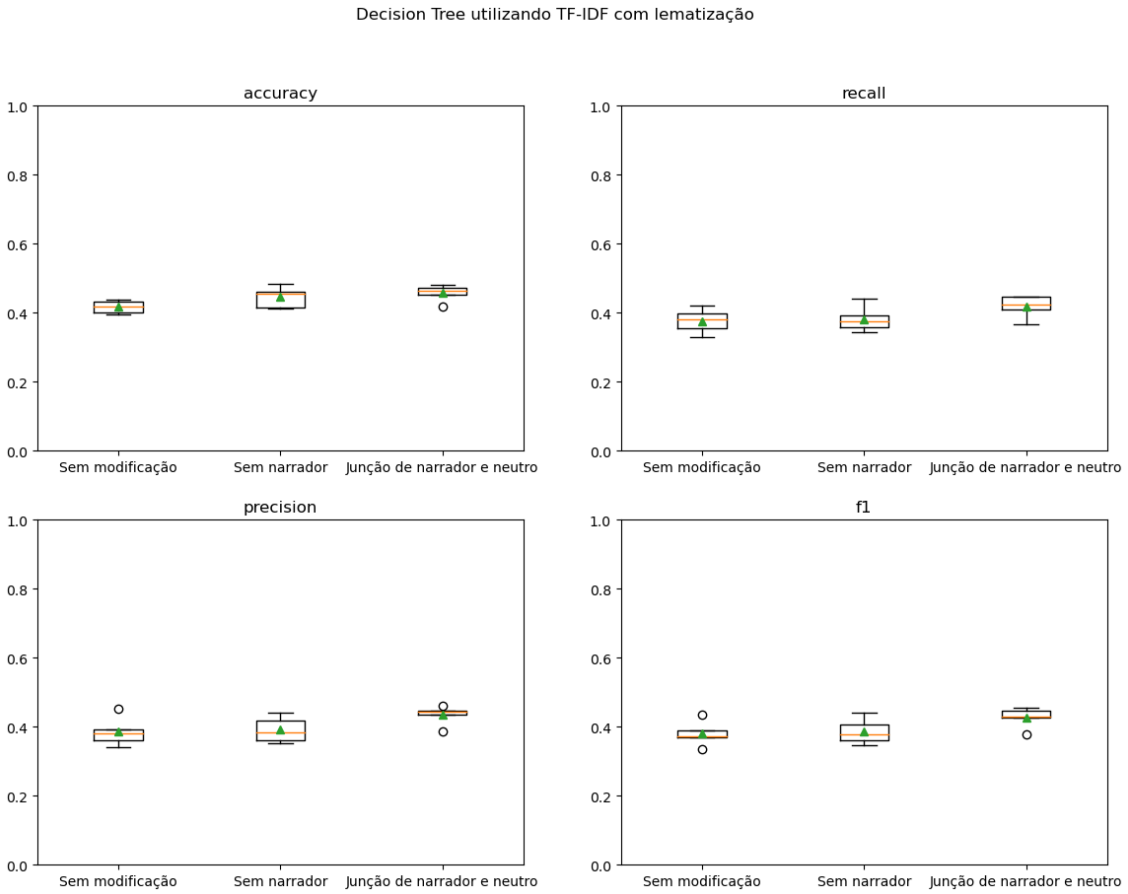

Figura 4. Métricas do modelo *decision tree* utilizando TF-IDF com lematização. Sendo, respectivamente, o primeiro gráfico com os valores de acurácia, o segundo com os valores de recall, o terceiro com os valores de precisão e o quarto com os valores de f1-score. No eixo X temos o tipo de processamento feito no atributo alvo e no eixo Y o valor da métrica variando na escala de [0, 1].

De forma semelhante à acurácia, vemos na Tabela 4 que não conseguimos encontrar *recall* médio acima de 50%. Tal qual verificamos na acurácia, os melhores resultados tendem a estar nos casos em que foram feitas alterações nas classes alvos. Também de

forma semelhante, observamos as árvores de decisões tendendo a ter os melhores valores, ainda que não se tenha nenhum efetivamente satisfatório.

Código do cenário	Sem modificação	Sem narrador	Junção de narrador e neutro
recall (C0)	39.59 ± 7.54	33.7 ± 0.89	40.96 ± 4.32
recall (C1)	37.62 ± 3.19	38.18 ± 3.4	41.92 ± 2.92
recall (C2)	31.9 ± 5.63	38.29 ± 2.95	40.58 ± 5.2
recall (C3)	29.2 ± 4.35	32.62 ± 2.29	36.46 ± 4.22
recall (C4)	24.19 ± 6.44	29.4 ± 2.32	31.55 ± 3.78
recall (C5)	27.38 ± 5.44	33.01 ± 5.18	36.83 ± 6.44
recall (C6)	21.74 ± 2.83	24.74 ± 0.7	26.45 ± 3.29
recall (C7)	30.13 ± 3.4	32.6 ± 0.59	34.3 ± 3.33
recall (C8)	14.86 ± 1.6	24.09 ± 3.56	19.9 ± 3.22

Tabela 4. Recall médio e desvio padrão de cada um dos cenários utilizados com a variação de classes alvo.

O único caso em que tivemos resultados médios acima de 50% foi na precisão. Foram encontrados baixos desvios padrões na utilização de KNN e *random forest*, ambos com matriz TF-IDF e sem lematização nos casos em que houve a junção de *narrator* e *neutral*. Em contrapartida nos três cenários que utilizamos *random forest* encontramos desvios padrões altos em uma das variações do atributo alto, de forma especifica no cenário em que não ocorreu a lematização na matriz TF-IDF e não houve alterações nas classes alvos. Nos outros dois cenários o resultado com alto desvio padrão foi encontrado na junção de *narrator* e *negative*.

Código do cenário	Sem modificação	Sem narrador	Junção de narrador e neutro
precisão (C0)	39.75 ± 4.73	33.51 ± 1.7	41.88 ± 4.2
precisão (C1)	38.52 ± 3.73	39.14 ± 3.37	43.51 ± 2.52
precisão (C2)	31.54 ± 5.52	38.74 ± 2.91	40.34 ± 5.17
precisão (C3)	46.36 ± 9.77	42.46 ± 6.53	53.41 ± 3.39
precisão (C4)	42.84 ± 6.86	37.15 ± 8.6	47.46 ± 4.86
precisão (C5)	40.18 ± 7.07	45.44 ± 8.97	48.52 ± 5.5
precisão (C6)	50.46 ± 18.15	40.07 ± 12.07	55.53 ± 2.19
precisão (C7)	46.83 ± 10.48	47.92 ± 14.05	54.2 ± 7.45
precisão (C8)	37.06 ± 12.73	38.83 ± 4.11	51.72 ± 12.33

Tabela 5. Precisão médio e desvio padrão de cada um dos cenários utilizados com a variação de classes alvo.

Análogo ao que aconteceu à acurácia e ao *recall*, vemos na Tabela 6, todos os valores médios de *f1* obtidos foram menores que 50%. Além disso, também alcançou-se melhor desempenho nos cenários em que houveram mudança nas classes alvo. Mesmo que nenhum valor tenha sido adequadamente satisfatório, na árvore de decisão observou-se melhores resultados. O KNN, em alguns casos, por exemplo, comparando os cenários C2 e C5, conseguiu resultados tão bons quanto, ou até melhores, que a árvore de decisão, embora com um desvio padrão maior.

Código do cenário	Sem modificação	Sem narrador	Junção de narrador e neutro
f1 (C0)	39.61 ± 6.07	33.57 ± 0.99	41.41 ± 4.23
f1 (C1)	38.03 ± 3.26	38.65 ± 3.35	42.69 ± 2.66
f1 (C2)	31.71 ± 5.57	38.51 ± 2.93	40.46 ± 5.18
f1 (C3)	35.6 ± 5.87	36.74 ± 3.47	43.22 ± 3.54
f1 (C4)	30.66 ± 6.52	32.59 ± 4.61	37.87 ± 4.14
f1 (C5)	32.38 ± 5.66	38.11 ± 6.36	41.78 ± 6.01
f1 (C6)	29.98 ± 5.93	29.97 ± 2.99	35.65 ± 2.92
f1 (C7)	36.43 ± 5.15	37.92 ± 4.3	41.85 ± 3.95
f1 (C8)	20.92 ± 3.51	29.69 ± 3.85	28.52 ± 4.85

Tabela 6. F1-score médio e desvio padrão de cada um dos cenários utilizados com a variação de classes alvo.

Por fim, dentre os classificadores tivemos o modelo de *random forest* com os melhores resultados em termos de precisão. Isto era esperado visto que o modelo utiliza internamente técnicas de *ensemble* com múltiplas árvores de decisão para predizer o resultado, tendendo, assim, a ter valores mais robustos. Isto se torna mais evidente tende em vista os dados de alta dimensionalidade que estamos usando – o que por si só já tende a dificultar a classificação –, sendo esse classificador o que melhor pode lidar com eles com suas técnicas de escolha de *features*.

Além disso, esse modelo foi o que mais se beneficiou da lematização. É possível notar que as métricas de acurácia, *recall*, *f1* e na precisão sem narrador, o cenário de TF-IDF com lematização se sobressaiu em relação aos outros dois. No caso da árvore de decisão, isso nem sempre acontece, e quando ocorre, a diferença no desempenho é menos expressiva. O modelo KNN foi o que menos se favoreceu da técnica, observando as Tabelas 3, 4, 5 e 6, com exceção da precisão sem modificações, o cenário TF-IDF com lematização alcançou os piores resultados.

6. Limitações

Como limitação temos que não foi utilizadas abordagens para a otimização dos hiperparâmetros dos algoritmos de aprendizado de máquina, sendo utilizados os padrões disponibilizados pela biblioteca scikit-learn. Isto prejudica o compreendimento se temos poucos dados para o aprendizado ou se estamos usando hiperparâmetros que não colaboram para este fim.

Outro ponto é a não utilização de técnicas próprias para redução de dimensionalidade. Ainda que tenhamos usado abordagens para que se tenha menos colunas, nas matrizes TF-IDF tivemos sempre mais do que 1000 atributos e nos vetores SBERT tivemos 512 atributos (tamanho máximo que o vetor pode ter). Logo, muitos dos resultados baixos que observamos podem ter sido causados por estas altas dimensionalidades e a utilização de técnicas apropriadas poderiam ter ajudado na tarefa de classificação.

Em relação aos classificadores, poderiam ter sido utilizadas outros que estão como estado da arte na área de Análise de Sentimentos, como um classificador BERT (*Bidirecional Encoder Representations from Transformers*), o qual tem apresentado um maior poder preditivo para muitas tarefas de PLN. Ainda que tenhamos utilizado uma

representação vetorial gerada a partir do SBERT, que é construído com redes BERT siamesas, não o utilizamos como classificador.

Por fim, poderiam ter sido feitas outras formas de lidar com a ambiguidade que alguns dados oferecem – mesma probabilidade para mais de um sentimento –, como a análise dos versos anteriores e/ou posteriores para escolher o sentimento do verso ambíguo. Outra abordagem é a utilização de ensemble das possíveis divisões por meio de classificadores treinados com cada uma das divisões, podendo ser todas possíveis ou as mais próximas encontradas, por uma abordagem de desambiguação.

Referências

Homero (2013). *Ilíada*. Penguin-Companhia. Tradução de Frederico Lourenço.

Pavlopoulos, J., Xenos, A., and Picca, D. (2022). Sentiment analysis of homeric text: The 1st book of iliad. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 7071–7077.

Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.