Universidade Federal de São João del Rei

Pró-Reitoria de Pesquisa e Graduação

CLASSIFICAÇÃO DE TEXTOS POR DISCIPLINA UTILIZANDO O FRAMEWORK TENSORFLOW E BERT

Relatório Final apresentado ao Programa Institucional de Bolsas de Iniciação Científica (PIBIC), para o Projeto AutorIA.

Autor: Mateus Gomes Martins Silva

Orientador: Fernando Augusto Teixeira

Curso: Engenharia Mecatrônica

Ouro Branco, 29 de março de 2022

RESUMO

Neste relatório será abordado o processo de construção de modelos de classificação, utilizando o framework Tensorflow e o modelo de atenção BERT. Serão apresentadas as etapas da construção, treinamento, teste e validação dos modelos de classificação, assim como breves considerações acerca da montagem de datasets e do processo de webscraping. Ao final serão apresentados os resultados obtidos e conclusões.

Palavras chave: webscraper. classificação, validação.

1. INTRODUÇÃO

Resumos acadêmicos sumarizam o conteúdo de um trabalho acadêmico ou artigo, e geralmente seguem um padrão de construções, utilizando expressões e sentenças semelhantes, independente da disciplina e tema em questão. O objetivo é construir um modelo que seja capaz de verificar se o resumo está coerente com tema abordado, basicamente o classificando com a disciplina correspondente. Foram utilizados dois tipos de classificação: binária, para classificar entre uma a duas disciplinas, e esparsa, para classificar mais de duas disciplinas.

1.1. JUSTIFICATIVA DA PESQUISA

Afim de executar uma abordagem prática e simplificada das possíveis funções do modelo de IA do AutorIA foi proposta a elaboração de um modelo de testes para classificação de texto, visando supervisionar se o resumo do aluno está dentro do contexto da disciplina, e do tema do trabalho acadêmico.

2. METODOLOGIA

Foram construídos 5 modelos Tensorflow, e um modelo BERT. Cada um utiliza formas diferentes de construção, variando em número de textos utilizados, disciplinas, épocas treinadas, neurônios, tipo e quantidade de layers utilizados. Embora haja diversas variações, a metodologia utilizada nos 5 modelos é a mesma, portanto serão apresentados os passos necessários para a construção geral, e as individualidades de cada um serão comentadas se necessário.

Linguagens, Bibliotecas e Frameworks utilizados:

- Linguagem utilizada: Python, Google Collab.
- Frameworks: Tensorflow, Keras, BERTimbau.
- Bibliotecas e módulos adicionais: NLTK, Beautiful Soup, Numpy, Pandas, Regex.

2.1. Modelos desenvolvidos

	Disciplinas	Qtd. Textos Treinamento	Qtd. Textos Teste
Modelo 1	Engenharia Elétrica Direito	74	8
Modelo 2	Engenharia Elétrica Direito	398	8
Modelo 3	Engenharia Elétrica Direito	398	8
Modelo 4	Eng. El. / Sistemas de Potência Eng. El. / Eletrônica e Automação.	820	8
Modelo 5	Geografia; Eng. Ambiental; Eng. El. Eletrônica; Eng. Elétrica; Direito; Odontologia; Eng. Mecânica; Computação.	1420	16
Modelo 6	Eng. El. / Sistemas de Potência Eng. El. / Eletrônica e Automação.	820	8

Tabela 1: Características dos modelos

2.2. Montagem do dataset

Para realizar o treinamento de um modelo de classificação, é necessário um dataset. O Dataset contém um conjunto de textos de cada disciplina, e sua devida classificação. Uma das dificuldades do projeto AutorIA era a falta de uma base de textos para testes, portanto para montar o dataset foi utilizado a técnica de Webscraping, ou seja, raspagem da web, onde um algoritmo busca e registra conteúdo de uma página ou série de páginas da web.

2.3. Webscraper

O *Webscraper* é o algoritmo responsável pela extração de conteúdo da web. Páginas de internet geralmente usam documentos HTML e/ou XML para formatação e armazenamento de dados necessários para sua exibição. Estas linguagens são chamadas de Markup Languages (linguagens de marcação), pois utilizam tags, ou marcadores para compor seu conteúdo. Utilizando requests e módulos como o Beautiful Soup, a técnica de webscraping consiste em encontrar determinado marcador ou id no documento, e executar comandos ou extrair conteúdo.

2.4. Definição da biblioteca

A Biblioteca Digital da USP foi escolhida para compor a base de dados a ser utilizada no Webscraper. Ao acessar determinado trabalho, a própria página contém um campo no qual o aluno deve colocar o resumo do documento. A página é um documento HTML, e contém um id, nomeado “DocumentoTextoResumo”. O webscraper basicamente extrai o conteúdo desse id, de todas as páginas da Biblioteca.

```
def html_to_txt_list(list_url):
    list_text = []
    for element in list_url:
        request = BeautifulSoup(requests.get(element).content,'lxml')
        text = ((request).find(id='DocumentoTextoResumo')).text
        list_text.append(text)
        print (list_text)
    return list_text

list_text_eletronica = html_to_txt_list(list_url_eletronica)
list_text_eletrica = html_to_txt_list(list_url_eletrica)
```

2.5. Criação do dataset e definição das classes

Para organizar as informações adquiridas pelo Webscraper, cada disciplina recebe uma identificação ou classe, que varia de acordo com o modelo. Para o Modelo de classificação binária, cada disciplina recebe um id de 0 ou 1. Para modelos de classificação esparsa, cada disciplina recebe um id de 1 a n, sendo n o número de disciplinas. As listas geradas pelo Webscraper são então salvas em um arquivo CSV, que será o *dataset* final. A primeira coluna sendo nomeada “Valor”, contém o valor numérico, ou *label*, que identifica a disciplina. A segunda coluna nomeada “Texto”, contém os resumos.

3. CONSTRUÇÃO, TESTE E VALIDAÇÃO DE UM MODELO

3.1. Carregar o dataset de treinamento:

São necessários dois datasets, um para treinamento, e outro para testes. Os textos devem ser diferentes, para evitar resultados falsos, baseados em memorização. O dataset de treinamento é carregado a partir do Framework Pandas, onde o primeiro passo é especificar a codificação utilizada, no caso UTF-8, e em seguida, realizar o embaralhamento dos dados. O dataset originalmente tem os textos organizados em

sequência, sendo necessário o embaralhamento para uniformizar a distribuição e evitar o desbalanceamento do modelo.

```
import pandas as pd
# leitura do arquivo CSV
dataset = pd.read_csv(nome_arquivo,encoding="utf-8")
# embaralhamento dos textos
dataset = dataset.sample(frac=1)
# transformação dos textos para o tipo String
dataset['texto'] = dataset['texto'].apply(str)
```

3.2. Limpeza do texto

Os textos obtidos pelo webscraper contém diversos erros, desde ortográficos à erros de formatação, causados pela conversão de html para txt. Alguns textos também contêm informações erradas, por exemplo, resumos em inglês ou espanhol, campos vazios, símbolos e outras falhas por parte do aluno.

É necessário realizar a limpeza do texto, para preparar as informações e criar o “saco de palavras”. Utilizando o NLTK, Regex, e funções padrões de *string* da linguagem Python, são retirados números, símbolos, URLs, e *stop words*, como artigos, conectivos e outros termos que não são necessários para o significado central da sentença. As palavras são colocadas todas em minúsculo.

```
from nltk.corpus import stopwords

# Lista de stop words em Português, disponível pelo pacote NLTK
stop_words = set(stopwords.words("portuguese"))

# Exemplo de função para remoção de stop words
def remove_stopwords(text):
    filtered_words = [word.lower() for word in text.split() if   word.lower() not in stop_words]
    return " ".join(filtered_words)
```

3.3. Word Counting:

Utilizando o módulo python Counter, é feita a contagem do número de palavras únicas nos textos do dataset, necessário para o processo de tokenização. Também são impressos os 5 termos mais usados nos textos, onde pode-se avaliar a semelhança entre

os resumos, independente da disciplina. Em alguns modelos de classificação, esses termos podem ser excluídos no processo de tokenização.

3.4. Dataset de treinamento e validação:

Para realizar o treinamento, é necessário dividir o dataset. Os dados são divididos em dois, sendo 80% para treinamento, e 20% para validação. Durante o treinamento, os dados de validação serão utilizados para calcular a acurácia do modelo.

```
# 80% dos dados para treinamento
tamanho_treinamento = int(dataset.shape[0]*0.8)
# lista com os primeiros 80%
dataset_treinamento = dataset[:tamanho_treinamento]
# lista com os últimos 20%
dataset_validacao = dataset[tamanho_treinamento:]
```

3.5. Tokenização:

A tokenização é o processo de transformar as palavras dos textos e frases em tokens, ou seja, os símbolos utilizados pelo modelo. Um texto é tratado como sentença. Ao tokenizar uma sentença, esta é transformada em uma sequência de inteiros.

O tokenizador do Tensorflow é capaz de retornar uma lista contendo todos os tokens criados com o dataset. Esta lista é o Word Index, que é utilizado para decodificar as sentenças tokenizadas. É possível atribuir valores de tokens pré-determinados, por exemplo, o token ‘<OOV>’, que identifica uma palavra que não está contida no index.

Em datasets prontos, como o IMDB utilizado em tutoriais Tensorflow, a formatação já está pronta para ser utilizada no modelo. Em datasets customizados, como o utilizado nos modelos em questão, é necessário transformar as sentenças e valores em arrays Numpy. Esta formatação é necessária para as operações que o tokenizador realiza internamente.

```
# Transformação das listas python em arrays Numpy

sentencas_treinamento = dataset_treinamento.texto.to_numpy()
labels_treinamento = dataset_treinamento.valor.to_numpy()

sentencas_validacao = dataset_validacao.texto.to_numpy()
labels_validacao = dataset_validacao.valor.to_numpy()
```

É realizada a tokenização dos *datasets* de treinamento e validação, onde são retornadas três listas, uma contendo as sequências de treinamento, outra com as sequencias de validação, e o *Word Index*. O Word index é formado apenas pelas palavras das sentenças de treinamento, ou seja, cada modelo tem seu próprio vocabulário.

```
from tensorflow.keras.preprocessing.text import Tokenizer

# Tokenizador carregado com o número de palavras do vocabulário
tokenizer = Tokenizer(num_words = num_palavras_unicas, oov_token = "<OOV>")
# A função fit_on_texts cria os tokens baseada nas sentenças de treinamento
tokenizer.fit_on_texts(sentencas_treinamento)
# Criação do Word Index
word_index = tokenizer.word_index
# Tokenização das sentenças de treinamento e validação
sequencias_treinamento = tokenizer.texts_to_sequences(sentencas_treinamento)
sequencias_validacao = tokenizer.texts_to_sequences(sentencas_validacao)
```

3.6. Padding:

Cada sentença tem um comprimento, ou seja, um número de palavras variado. Para o treinamento, é necessário que este comprimento seja padronizado para todas as sequências, e para isso é utilizado o processo de *padding*, onde é definido o número máximo de tokens em uma sequência, e o tipo de padding. No modelo em questão é utilizado o tipo "*post*". Este processo de normaliza os arrays de sequências retornados pelo tokenizador, para que todas as sentenças tenham o mesmo comprimento. Para ter noção de qual deve ser o média do comprimento das sentenças, pode-se utilizar gráficos de distribuição de frequência, com o módulo *Seaborn*, como visto na Figura 1.

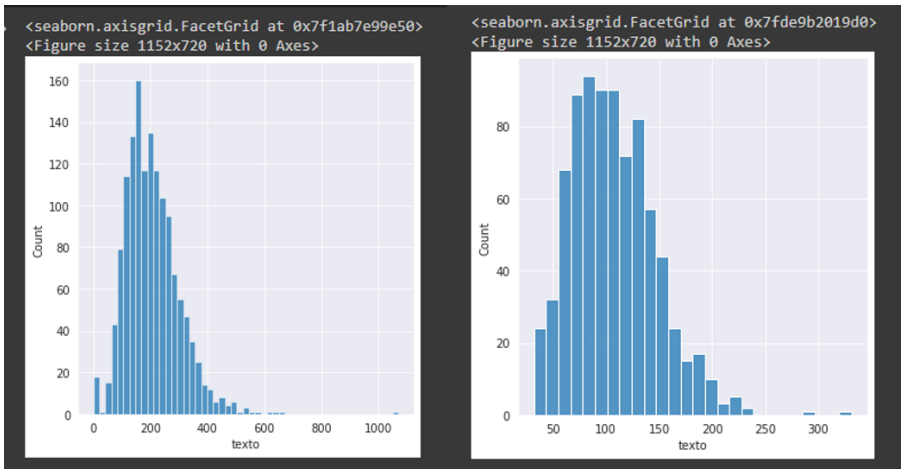

Figura 1 Distribuição de Frequência com Seaborn. Fonte: Autor.

No tipo "post", caso o número de tokens seja maior que o especificado, uma nova lista é retornada contendo apenas os primeiros tokens, e os excedentes são excluídos. Caso a lista seja menor, novos tokens de valor zero são adicionados até atingir o valor especificado.

```
  from tensorflow.keras.preprocessing.sequence import pad_sequences

  num_max_palavras = 100 # Deve ser menor que 999
  # Criaçao das listas de sequencias com comprimento padronizado
  padded_treinamento = pad_sequences(sequencias_treinamento, maxlen =
num_max_palavras, padding = "post",truncating = "post")
  padded_validacao = pad_sequences(sequencias_validacao, maxlen =   num_max_palavras,
padding = "post", truncating = "post")
```

3.7. MODELO TENSORFLOW:

3.7.1. Modelo sequencial e input Layers

Primeiramente é criado um modelo Sequencial, onde os *layers* são agrupados sequencialmente, como uma pilha. Cada layer é responsável por uma operação distinta com relação aos dados, e contém diversos parâmetros para refinar a operação.

Neste modelo de classificação, o primeiro layer é uma camada de *Embedding*, onde as sequências de tokens criadas anteriormente serão transformadas em vetores, cada um com um valor fixo. Recebe geralmente 3 parâmetros, sendo eles:

- O número de palavras únicas do dataset.
- O tamanho do batch, ou pacote de palavras.
- O tamanho da sequência, ou o número de palavras por sentença.

O próximo *layer* definirá quais as operações matemáticas utilizadas para tratar os batches criados na camada de Embedding. Existem diversas classes disponíveis, mas a utilizada nestes modelos foi GlobalAveragePooling1D, que não recebeu parâmetros.

```
  from tensorflow.keras import layers
  num_palavras_unicas = 1000   # tamanho do vocabulário
  batch_size = 32              # tamanho do pacote de palavras
  model = keras.models.Sequential()  # Modelo Sequencial
  model.add(layers.Embedding(num_palavras_unicas, batch_size, input_length =
num_max_palavras))
  model.add(layers.GlobalAveragePooling1D())
```

3.7.2. Hidden layers

Os próximos layers são layers densos, que aumentam a complexidade da rede neural, melhorando o desempenho do modelo, e criando forma para a saída dos resultados. Não tem número definido, podem ser adicionados ou subtraídos, de acordo com a resposta do treinamento. Recebem geralmente 2 parâmetros, sendo eles:

- Número de neurônios da camada.
- Função de ativação (Relu, Softmax, Sigmoid, etc).

```
model.add(layers.Dense(128, activation = "relu"))
model.add(layers.Dense(24, activation = "sigmoid"))
```

3.7.3. Output layer e função loss

O último Layer é o que define a saída dos resultados, ou seja, os neurônios que irão retornar a classificação do texto analisado pelo modelo. O número de neurônios e a função de ativação vão depender do tipo de classificação, e da quantidade de valores, ou labels, definidos no Dataset.

O tipo de classificação é definido pela Função Loss. Resumidamente, a função Loss retorna um erro estatístico, que compara os resultados do último treinamento com o anterior. Desta forma, quanto menor for o valor da Função Loss, mais preciso será o modelo. De fato, o processo de melhoria de um modelo se resume a reduzir o valor de loss. Existem diversas classes para a função Loss, mas nos modelos em questão foram adotados 2 tipos:

- Binary Crossentropy
- Sparse Catergorical Crossentropy

Em um dataset que contém textos de Matemática e Geografia, haverá 2 valores, ou labels, um para cada disciplina. Supondo que os valores sejam 0 (zero) para Matemática e 1 (um) para Geografia, o modelo pode ser moldado de duas formas diferentes.

Utilizando a função Loss Binary Crossentropy, cria-se um modelo de classificação binária, que irá retornar 0 ou 1, de acordo com o texto avaliado. Esse modelo é treinado com datasets de um ou dois valores, e seu último Layer será denso, com um neurônio. Pode-se pensar na resposta de um modelo binário como uma resposta "sim" ou "não".

Utilizando a função Loss Sparse Categorical Crossentropy, pode-se treinar datasets com diversos valores, por exemplo o Modelo 5, o qual contém 8 disciplinas diferentes, terá como último Layer um Layer denso, de 9 neurônios, (a classificação ocorre de 0 a 8). A função de ativação utilizada foi a "SoftMax".

```
# Para classificação Binária – Modelo 1
model.add(layers.Dense(2, activation = "sigmoid"))

# Para classificação Categórica Esparsa – Modelo 5
model.add(layers.Dense(9, activation = "softmax"))

model.summary()   # Finalização dos Layers
```

A função SoftMax retorna um array com valores de probabilidade. A finalização dos Layers é realizada pela função “model.summary()”. O resultado pode ser visto na Figura 2:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 250, 32)           376736
_________________________________________________________________
global_average_pooling1d (Gl (None, 32)                0
_________________________________________________________________
dense (Dense)                (None, 128)               4224
_________________________________________________________________
dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_2 (Dense)              (None, 24)                3096
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 50
=================================================================
Total params: 400,618
Trainable params: 400,618
Non-trainable params: 0
```

Figura 2 Saída no Terminal

3.7.4. Compilação do modelo

Definidos os Layers, é realizada a compilação. A compilação de um Modelo Tensorflow define as configurações para o treinamento. Nos modelos em questão foram definidos três parâmetros, sendo eles:

- Função Loss, definida de acordo com o Output Layer.
- Otimizador, sendo o "Adam" para todos os modelos (Learning Rate = 0.001).
- Métrica, ou a medida para a analisar os resultados do modelo. Foi utilizada “*Acuraccy*” (Precisão) para todos os modelos.

```
loss = keras.losses.SparseCategoricalCrossentropy(from_logits=False)
optim = keras.optimizers.Adam(lr=0.001)
metrics = ["accuracy"]

model.compile(loss=loss, optimizer = optim, metrics = metrics)
```

3.7.5. Treinamento e validação

Definidos todos os parâmetros e configurações do modelo, realiza-se o treinamento utilizando a função “model.fit()”. São passados cinco parâmetros, sendo eles:

- Array com as sequências de treinamento.
- Array com os valores (labels) de treinamento.
- Número de Épocas. Varia para cada modelo.
- dddddDados de Validação, uma tupla contendo os arrays das sequências e dos valores de validação.
- Verbose, ou a forma de visualizar o progresso durante o treinamento. Foi usado "2" em todos os modelos.

```
history = model.fit(padded_treinamento, labels_treinamento, epochs = 10,
validation_data = (padded_validacao, labels_validacao), verbose=2)
```

Pode-se observar na Figura 3, a variação do valor de Loss, em relação à precisão do modelo. Quanto mais preciso o modelo, menor o valor da função Loss.

```
21/21 - 0s - loss: 0.6186 - accuracy: 0.7881 - val_loss: 0.6040 - val_accuracy: 0.7561
Epoch 5/10
21/21 - 0s - loss: 0.4012 - accuracy: 0.8750 - val_loss: 0.4896 - val_accuracy: 0.7805
Epoch 6/10
21/21 - 0s - loss: 0.1600 - accuracy: 0.9558 - val_loss: 0.4886 - val_accuracy: 0.7988
Epoch 7/10
21/21 - 0s - loss: 0.0547 - accuracy: 0.9909 - val_loss: 0.5610 - val_accuracy: 0.7866
Epoch 8/10
21/21 - 0s - loss: 0.0214 - accuracy: 0.9970 - val_loss: 0.5931 - val_accuracy: 0.7866
Epoch 9/10
21/21 - 0s - loss: 0.0085 - accuracy: 0.9985 - val_loss: 0.6542 - val_accuracy: 0.7866
Epoch 10/10
21/21 - 0s - loss: 0.0054 - accuracy: 1.0000 - val_loss: 0.6849 - val_accuracy: 0.7683

Finished in 4.0 second(s)
```

Figura 3 Treinamento Tensorflow. Fonte: Autor.

Utilizando a variável “history”, pode-se gerar gráficos que mostram a variação dos parâmetros de treinamento e loss em função do tempo.

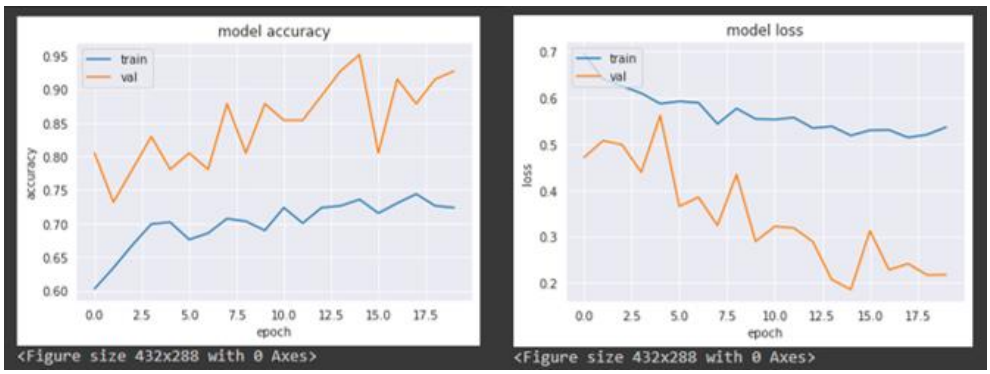

Figura 4 Variação de Precisão e Loss no tempo. Fonte: Autor.

3.7.6. CARREGAR E PREPARAR DATASET DE TESTE

Com o modelo treinado, se realiza o teste com textos de um dataset composto por textos que o modelo não "viu" durante o treinamento. Sentenças diferentes, comprimentos diferentes, e provavelmente novas palavras. O dataset de teste mantém a mesma organização do de treinamento, ou seja, uma coluna de valores, ou labels, e uma com os resumos. Utilizar um dataset acelera o processo de testes do modelo, em comparação com sentenças isoladas (textos avulsos).

Este dataset, assim como qualquer sentença deve passar pela mesma preparação do treinamento, ou seja, embaralhamento dos dados, limpeza do texto, conversão para array numpy e tokenização.

```
sequencias_teste = tokenizer.texts_to_sequences(sentencas_teste)
padded_teste = pad_sequences(sequencias_teste, maxlen = num_max_palavras,
                             padding = "post",truncating = "post")
```

3.7.7. TESTES COM O DATASET DE TESTES

Para realizar o teste, é utilizada a função “model.predict()”, que recebe como parâmetro as sequencias preparadas anteriormente. Em modelos de Classificação Binária, ela retornará uma lista com valores entre 0 e 1. Basta considerar 0 para valores menores ou iguais a 0.5 e 1 para valores maiores que 0.5.

Em modelos de Classificação Categórica Esparsa, ela retornará uma lista de probabilidades para cada sequência. Para retornar o label correspondente, se usa a função Numpy “np.argmax()”.

```
# previsoes geradas com o modelo
previsoes = model.predict(padded_teste)

# Para Classificação Binária se utiliza:
 previsoes = [1 if p > 0.5 else 0 for p in previsoes]

# Para Classificação Categórica Esparsa se utiliza:
previsoes = [np.argmax(element) for element in previsoes]

print(labels_teste) # impressão dos labels originais para comparação
print(previsoes)    # impressão dos labels previstos pelo modelo
```

Pode-se observar na Figura 5 que o modelo previu corretamente a disciplina de 7 dos textos do dataset de teste. Isso aponta uma precisão de 87.5%, maior que os 76% observados durante o treinamento.

```
[1 1 1 1 0 0 0 0]
[1, 1, 0, 1, 0, 0, 0, 0]
```

Figura 5 Previsão do Modelo. Fonte: Autor.

3.7.8. SALVAR E CARREGAR UM MODELO

Concluído o treinamento e testes, o modelo e o tokenizador devem ser salvos, para serem utilizados na aplicação de fato, ou testes. Pode-se melhorar a acurácia e expandir o vocabulário através de novos treinamentos. O tokenizador pode ser salvo como um arquivo JSON, e irá manter o vocabulário utilizado no treinamento.

```
model.save('./saved_models') # Diretório do Arquivo
import json
json_string = tokenizer.to_json() # O tokenizador é salvo como um arquivo JSON
with open('tokenizer.json','w') as arquivo:
  json.dump(json_string, arquivo)
```

3.8. USANDO BERT PARA CLASSIFICAÇÃO

BERTimbal é um modelo BERT pré treinado para a língua portuguesa, desenvolvido pela NeuralMind AI. Em modelos de Classificação, pode-se utilizar a capacidade de *Enconding* multidirecional e máscaras do BERT para aperfeiçoar a predição dos resultados. O encoding multidirecional permite que o significado da palavra seja dado pelo contexto, e não apenas pela sentença.

O modelo de classificação utilizando o Bert diverge do Modelo Tensorflow padrão a partir da tokenização.

3.8.1. TOKENIZAÇÃO, PADDING E MÁSCARAS

Foi utilizado o BERT Base, por ser mais leve, mas suficiente para a tarefa de classificação. No Tensorflow puro, a tokenização e o encondig são processos manuais, ou seja, é necessário realizar a conversão do dataset em arrays numpy, realizar a tokenização, definir os caracteres especiais e realizar o processo de padding. O tokenizador do BERT implementa todas estas funções, retornando não apenas a lista de Tokens, chamada no BERT de IDs, mas também a lista das máscaras.

```
  from transformers import AutoTokenizer
  tokenizer = AutoTokenizer.from_pretrained('neuralmind/bert-base-portuguese-
cased') # Modelo BERT para a língua portuguesa

  Xids = np.zeros((len(df),SEQ_LEN))  # Inicialização das matrizes que ...
  Xmask = np.zeros((len(df),SEQ_LEN)) # ... receberão os IDs e as Máscaras

  for i, sequence in enumerate(df['texto']):
      tokens = tokenizer.encode_plus(sequence, max_length=SEQ_LEN,
                                     truncation= True, padding="max_length",
                                     add_special_tokens= True,
                                     return_token_type_ids= False,
                                     return_attention_mask= True,
                                     return_tensors= 'tf')
      Xids[i,:], Xmask[i,:] = tokens['input_ids'], tokens['attention_mask']
```

Com os tokens, mascaras, e labels, monta-se o dataset transformado, necessário para treinamento do modelo BERT. É realizado um novo embaralhamento, e o dataset é dividido entre treinamento e validação:

```
  dataset = tf.data.Dataset.from_tensor_slices((Xids,Xmask,labels))

  def map_function(input_ids,mask,labels):
    return {'input_ids':input_ids, 'attention_mask':mask}, labels

  dataset = dataset.map(map_function) # Masks e Ids recebem um nome
  dataset = dataset.shuffle(10000).batch(2) # Embaralhamento e batches de 2 palavras
```

3.8.2. MODELO SEQUENCIAL E INPUT LAYERS

O Modelo Bert é o mesmo utilizado no tokenizador. O Input Layer recebe a lista de Ids, Mascaras e labels geradas pelo tokenizador. O layer de Embedding irá criar o encoding multidirecional para as sentenças, diferente dos vetores de valor fixo gerado nos modelos Tensorflow.

```
  from transformers import TFAutoModel
  bert = TFAutoModel.from_pretrained('neuralmind/bert-base-portuguese-cased',  from_pt=
True)
   input_ids = tf.keras.layers.Input(shape=(SEQ_LEN,),name='input_ids',  dtype='int32')
  mask = tf.keras.layers.Input(shape=(SEQ_LEN,),name='attention_mask', dtype='int32')
  embeddings = bert(input_ids, attention_mask = mask)[0]
```

3.8.3. OUTPUT LAYER E FUNÇÃO LOSS

Os layers densos de um modelo BERT são idênticos aos de um modelo Tensorflow, a diferença é que o BERT contém Layers que podem ser retreinados. A base de parâmetros do BERT é extensa, portanto, é necessário informar que não sejam adicionados.

```
Y = tf.keras.layers.Dense(2,activation='softmax',name='outputs')(X)
model = tf.keras.Model(inputs=[input_ids,mask],outputs=Y)

model.layers[2].trainable = False # Necessário para que o BERT nao seja retreinado
model.summary()
```

3.8.4. COMPILAÇÃO, TREINAMENTO E VALIDAÇÃO

Estes passos são idênticos ao modelo Tensorflow, mas há diferenças na curva de aprendizado, que demostra uma boa precisão.

Figura 6 Treinamento Modelo TF.BERT. Fonte: Autor.

3.8.5. TESTES COM O DATASET DE TESTES:

O dataset de testes é carregado da mesma forma que o modelo tensorflow, passando pelo processo de remoção de stop words, símbolos, urls, tokenização, sequenciamento e padding. Aplicando a função de previsão, se obtém no terminal a seguinte saída:

```
[0, 1, 1, 1, 1, 0, 0, 0]
100.0
```

Figura 7 Teste Modelo TF.BERT. Fonte: Autor

O modelo BERT previu 100% dos textos do dataset de testes

4. RESULTADOS

4.1. Modelos Tensorflow

Na tabela abaixo pode-se observar a média do tempo de treinamento, valores de Loss e Precisão para cada modelo. Os modelos foram treinados no Google Collab, na configuração padrão, e foram realizados 5 testes para cada.

	Tempo (s)	Loss	Acuraccy	Val Loss	Val Accuracy
Modelo 1	19,12	0,2020	0,9572	0,5530	0,8667
Modelo 2	84,04	0,0955	0,9704	0,5093	0,8875
Modelo 3	4,60	0,4068	1,0000	0,4408	0,9550
Modelo 4	5,56	0,0016	1,0000	0,9070	0,7866
Modelo 5	12,70	0,2188	0,9637	0,7273	0,7451

Tabela 2: Comparativo Modelos Tensorflow. Fonte: Autor.

Pode-se observar o tempo excessivo de treinamento dos primeiros modelos, mesmo utilizando os menores datasets. Isto se deve ao layer de operações Matemáticas, no caso LSTM (Long Short Term Memory). Este layer se mostrou lento, e funciona bem para sentenças de até 100 palavras. Requer grandes datasets para atingir precisão expressiva, portanto não foi adequado. Foi substituído pelo Layer Global Average Pooling 1D, mais rápido e mais preciso para pequenos datasets.

Alguns modelos tinham boa precisão, mas apresentavam grande valor de Loss, como o modelo 1 e 3. Isto significa que ele apenas “memorizou” o contéudo.

Os modelos 4 e 5 foram os que apresentaram os resultados mais satisfatórios, pois contavam com grandes datasets, e modelos complexos. Embora não sejam tão precisos na validação, ambos atingiram entre 87,5 a 92,5% de acerto com o dataset de testes.

4.2. Modelo BERTimbau

A tabela abaixo exibe a comparação entre dois modelos BERT, idênticos com relação ao dataset e tokenização, mas com diferenças no layer de operação matemática. O Modelo 1 foi treinado com GlobalMaxPool1D em 50 épocas, e o Modelo 2 com LSTM em 20 épocas.

	Tempo (s)	Loss	Acuraccy	Val Loss	Val Accuracy
Modelo 1	2082,58	0,4403	0,8022	0,0728	0,9878
Modelo 2	1540,92	0,0908	0,9603	0,5093	0,8875

Embora o modelo 1 tenha levado mais tempo, devido ao grande número de épocas, o *decay* da função loss, ou seja, sua taxa de aceleração foi baixa, oque possibilitou um modelo melhor construído. O modelo 2 é um perfeito exemplo de *over training*, ou seja, o modelo perde a referência à medida que aumenta, como mostra a curva (Figura 8):

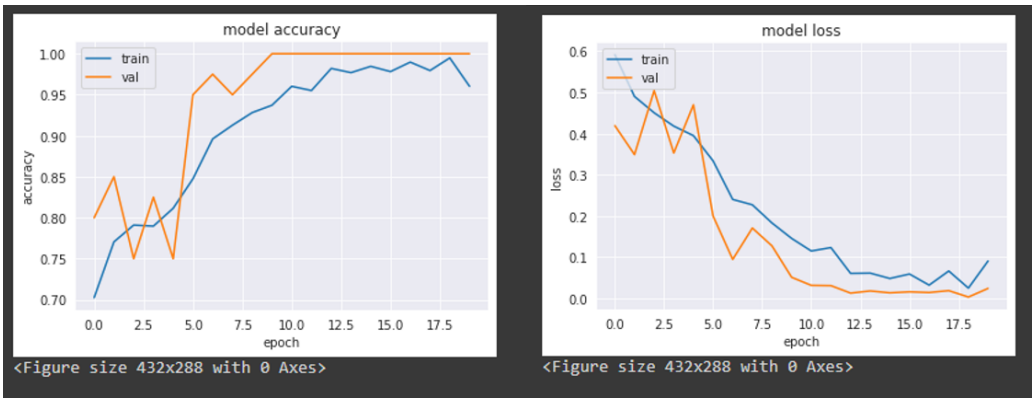

Figura 8 Treinamento TF.BERT 2. Fonte: Autor.

Embora a precisão seja maior que o modelo 1, o desempenho com o dataset de testes foi muito abaixo do esperado, acertando apenas 75%:

```
[0, 0, 0, 1, 1, 0, 0, 0]
75.0
```

Figura 9 Teste Modelo TF.BERT 2. Fonte: Autor.

4.3. Conclusão

O BERT se mostrou o modelo mais preciso, mas tem certas desvantagens, sendo a primeira o tempo de treinamento, e a segunda o tamanho do modelo. Ocupa muito espaço em disco, oque pode não ser interessante para o AutorIA via Web. Já o modelo Tensorflow tem como vantagens a rapidez do treinamento, e é um modelo leve, tanto em tamanho quanto em consumo de recursos.

4.4. Agradecimentos

Agradeço a UFSJ pela possibilidade de fazer parte da pesquisa científica, e pelo suporte financeiro e educacional durante o projeto.

Agradeço ao Professor Fernando Augusto Teixeira pelas orientações e aprendizado acumulado durante o projeto.

REFERÊNCIA BIBLIOGRÁFICA

ANALYTICS VIDHYA. **Guide for Loss Function in Tensorflow**, Pravin Borate. Disponível em: <https://www.analyticsvidhya.com/blog/2021/05/guide-for-loss-function-in-tensorflow/> Acesso em 20/07/2021.

PYTHON ENGINNER. **Text Classification in Tensorflow**. Disponível em: <https://github.com/python-engineer/tensorflow-course> Acesso em 20/07/2021.

TENSORFLOW. **Tf.keras.layers. Public API for tf.keras Layers**. Disponível em: <https://www.tensorflow.org/api_docs/python/tf/keras/layers> Acesso em 20/07/2021.

TENSORFLOW. **Embeddings de Palavras.** Disponível em: <https://www.tensorflow.org/tutorials/text/word_embeddings> Acesso em 20/07/2021.

TENSORFLOW. **Classificação de Texto com Bert.** Disponível em: <https://www.tensorflow.org/text/tutorials/classify_text_with_bert> Acesso em 15/10/2021.

BETTER PROGRAMMING. **Build a Natural Language Classifier With Bert and Tensorflow**. Disponível em: <https://betterprogramming.pub/build-a-natural-language-classifier-with-bert-and-tensorflow-4770d4442d41> Acesso em 15/10/2021.