Natural Language and Information Retrieval
Degree in Data Science
ETSINF. Academic year 2022-2023

Practical Work 3

Session 5: BETO and RoBERTa (Spanish) for text classification tasks.

Reynier Ortega Bueno
rortega@prhlt.upv.es

Goal

The goal of this lab session is to help students understand and gain practice in the use of deep learning-based language models, in particular transformer-based models (BERT, BETO and RoBERTa). The second objective is the application of these models to text classification tasks, particularly for the HUHU shared task.

Bibliography

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. Advances in neural information processing systems, 30.
- Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. NAACL-HLT 2019, pages 4171–4186, ACL.
- Cañete, J., Chaperon, G., Fuentes, R., Ho, J. H., Kang, H., & Pérez, J. (2020). Spanish pre-trained bert model and evaluation data. In Proceeding of the Practical ML for Developing Countries Workshop @ ICLR 2020, pages 1-9, Addis Ababa, Ethiopia.
- De la Rosa, J., Ponferrada, E., Romero, M., Villegas, P., González de Prado Salas, P., & Grandury, M. (2022). BERTIN: Efficient Pre-Training of a Spanish Language Model using Perplexity Sampling. Procesamiento Del Lenguaje Natural, 68, 13-23
- Cañete, J., Donoso, S., Bravo-Marquez, F., Carvallo, A., & Araujo, V. (2022). Albeto and distilbeto: Lightweight spanish language models. arXiv preprint arXiv:2204.09145.

- Gutiérrez-Fandiño, A., Armengol-Estapé, J., Pàmies, M., Llop-Palao, J., Silveira-Ocampo, J., Pio Carrino, C., Armentano-Oller, C., Rodriguez-Penagos, C., Gonzalez-Agirre, A., & Villegas, M. (2022). MarIA: Spanish Language Models. Procesamiento Del Lenguaje Natural, 68, 39-60.
- https://huggingface.co/docs/transformers/index

BERT

BERT is described as one of the most significant breakthroughs in the history of (Google's) Search [3]. Specifically, it is used in the disambiguation of queries in all searches in English, but other languages are also included, such as Spanish, Portuguese, Hindi, Arabic, German, etc. [4]. BERT has also been used in many automatic text classification tasks and other Natural Language Processing tasks.

The general idea of building a BERT-based classifier consists of two essential stages, as seen in Figure 1. The first is to obtain the (deep) representation of the texts from the input data and use said representation as input to a classifier, which is usually an MLP that receives the output of the BERT model as input and outputs a layer with as many neurons as classes in the task.

It is essential to highlight that the BERT model can be used directly from the general data with which it was trained (pre-trained), or it can be fine-tuned to consider specific aspects of the task to be solved (fine-tuned).

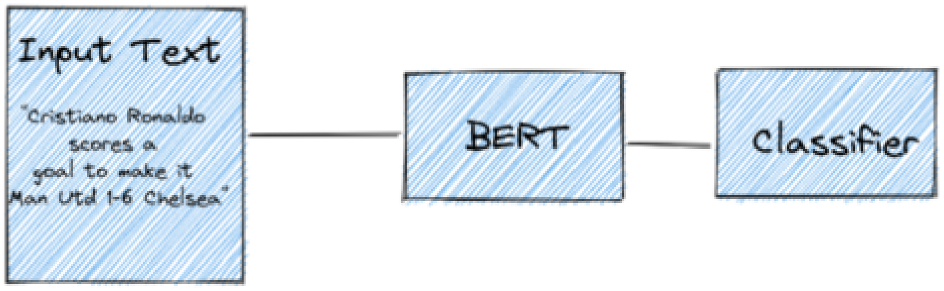
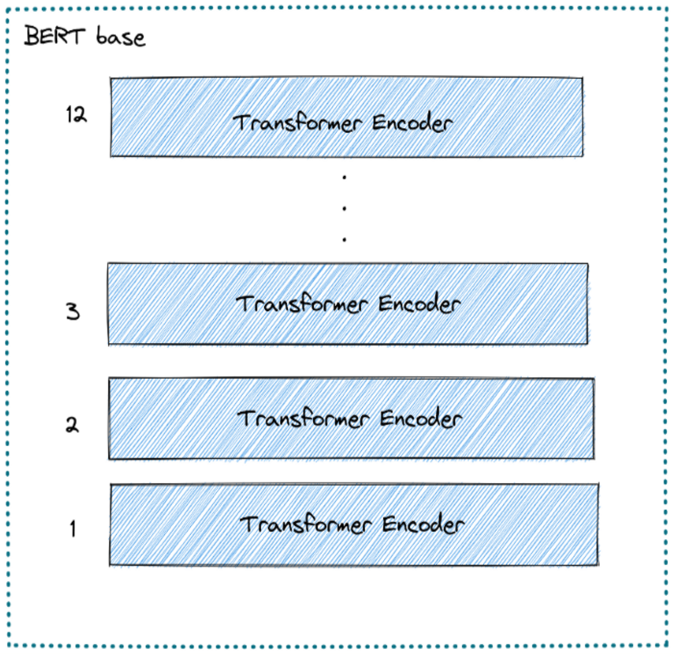

Input Text
"Cristiano Ronaldo scores a goal to make it Man Utd 1-6 Chelsea"
BERT
Classifier

Fig 1. A general architecture for using BERT in classification tasks

BERT is an acronym for Bidirectional Encoder Representations from Transformers. The name itself provides clues as to what the BERT model consists of. The BERT architecture consists of stacked encoders (see Figure 2). Each encoder encapsulates two sublayers: a self-attention layer and a feed-forward layer (see Figure 3).



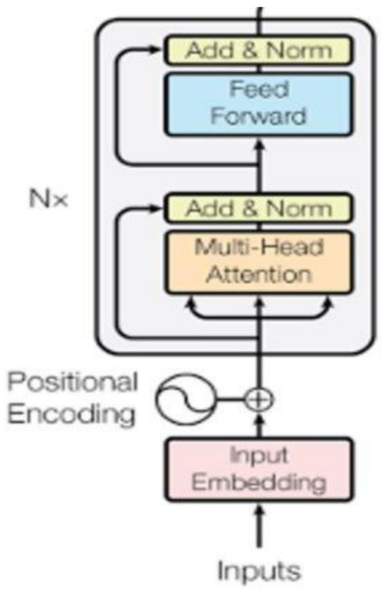


Fig 2. BERT architecture

Fig. 3 Transformer Econder

There are different models of BERT regarding the size of the architecture:

- Base BERT is a model consisting of 12 encoder layers, 12 self-attention heads, 768 hidden sizes, and 110 million parameters.

- Large BERT is a model consisting of 24 encoder layers, 16 self-attention heads, 1024 hidden sizes, and 340 million parameters.

There are two reasons why BERT is a robust language model:

1. It is pre-trained with unlabeled data from BooksCorpus (800 million words) and Wikipedia (2.5 billion words).
2. It is pre-trained using the bidirectional nature of stacked encoders. This fact means that BERT learns the information in a sequence of words from left to right and from right to left.

How does BERT process the input data?

Given the sentence below, the first step would be to get the tokens (words) of the sequence.

"I will watch memento tonight"

The BERT model requires a sequence of tokens (words) as its input. Within each token sequence, there are two special tokens that must be included according to BERT's expectations (see Figure 4):

- [CLS]: This token marks the beginning of each sequence and is known as the classification token.

- [SEP]: This token serves the purpose of informing BERT about the boundaries between sentences within the token sequence. It plays a crucial role, especially in tasks involving next sentence prediction or question answering. In cases where there is only one sentence, this token is added at the end of the sequence.

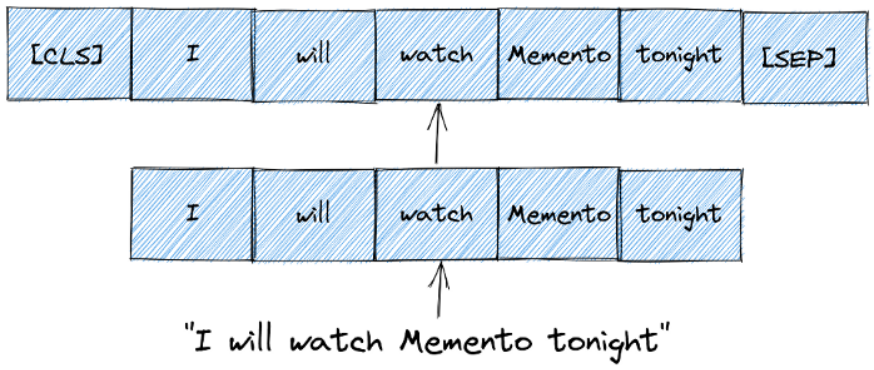


Fig 4. Tokenized textual input

Transforming the input sentence into a token sequence for BERT requires just a single line of code. To achieve this, we utilize the BertTokenizer class, which will be demonstrated later. It is crucial to consider that the maximum allowable token length for the BERT model is 512. If a sequence's tokens fall short of this limit, sequence completion with the [PAD] token can be applied. On the other hand, if the token sequence surpasses the 512-token limit, truncation is performed accordingly.

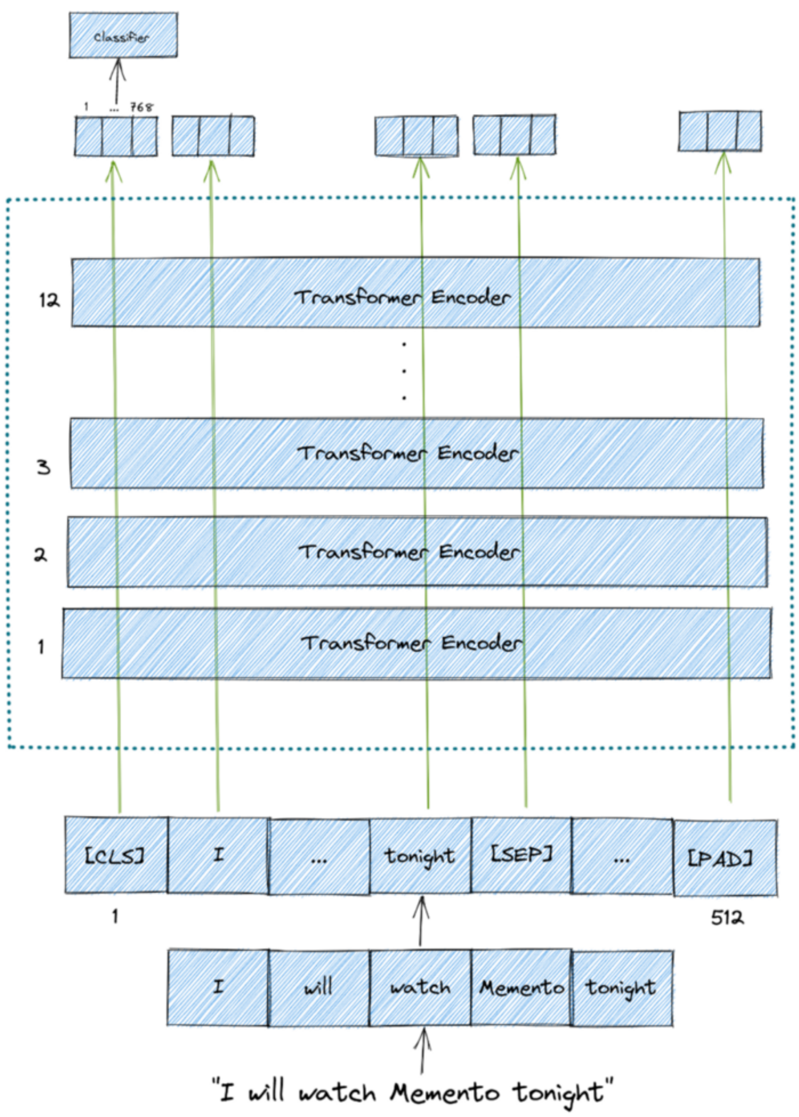


Fig 5. BERT output for a textual input.

The BERT model then generates a 768-dimensional embedding vector for each token. These vectors can serve as input for various Natural Language Processing applications, including text classification, next sentence prediction, Named Entity Recognition (NER), and Question & Answering.

In the case of text classification, we focus on the embedding vector output of the special token [CLS]. This entails utilizing the 768-dimensional embedding vector of the [CLS] token as input for the classifier (MLP). The classifier, in turn, generates a vector of a size corresponding to the number of classes in the classification task. This process is illustrated in Figure 5.

BETO and RoBERTa (for Spanish)

BETO is an initiative that enables the use of the pre-trained BERT model for Spanish language. BETO has similar size to the BERT-base model. In this case, a model with 12 encoders and 16 self-attention heads was utilized, with a hidden layer size of 1024. The model consists of a total of 110 million parameters. For training the model, texts were gathered from various sources, including Wikipedia and Spanish-language sources from the OPUS Project. These sources encompass United Nations and Government journals, TED Talks, subtitles, news articles, and more. The overall corpus size is comparable to the corpus used in training the original BERT model.

RoBERTA Spanish is a transformer-based masked language model for the Spanish language. It is based on the RoBERTa base model and has been pre-trained using the largest Spanish corpus known to date, with a total of 570GB of clean and deduplicated text processed for this work, compiled from the web crawlings performed by the National Library of Spain from 2009 to 2019.

USING PRE-TRAINED BETO

The first step is to load the pre-trained model for the BETO tokenizer. This can be done using the transformers library in Python and the pre-trained models available in HuggingFace. Specifically, the pre-trained model for BETO can be used by employing the name *'dccuchile/bert-base-spanish-wwm-uncased'.*

Tokenization with BERT Tokenizer

EXAMPLE 1

```
!pip install transformers
import torch
from transformers import BertTokenizer
tokenizer= BertTokenizer.from_pretrained('dccuchile/bert-base-spanish-wwm-uncased')
#The max_length parameter depends on the  texts' length in the dataset
#Examples of the tokenization performed by BERT·
example_text  =  ['Usando  BETO  en  clases  de  ciencia  de  datos  de  la  universidad
politécnica.', "Los estudiantes de este grado son muy aplicados y estudiosos."]
tokenized_text = tokenizer.tokenize(example_text[0])
print(tokenized_text)
tokenized_text = tokenizer.tokenize(example_text[1])
print(tokenized_text)
#X[:100].to_list()
bert_input = tokenizer(example_text,padding='max_length', max_length = 20,
                      truncation=False, return_tensors="pt")

# Transform tokens to vocabulary indexes
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
#BERT WORKS WITH THESE REPRESENTATIONS OBTAINED FOR THE TOKENS
print(bert_input['input_ids'])
print(bert_input['token_type_ids'])
print(bert_input['attention_mask'])
#To transform the sequence of token indices to words in the texts
example_text = tokenizer.decode(bert_input.input_ids[0])
print(example_text)
example_text = tokenizer.decode(bert_input.input_ids[1])
print(example_text)
```

USING PRE-TRAINED RoBERTa for Spanish

The first step is to load the pre-trained model for the RoBERTa tokenizer. This can be done using the transformers library in Python and the pre-trained models available in HuggingFace. Specifically, the pre-trained model for RoBERTa for Spanish can be used by employing the name *"PlanTL-GOB-ES/roberta-base-bne".*

Tokenization with RoBERTa Tokenizer

EXAMPLE 2

```
import torch
from transformers import RobertaTokenizer, RobertaModel
#The max_length parameter depends on the  texts' length in the dataset
tokenizer= RobertaTokenizer.from_pretrained('PlanTL-GOB-ES/roberta-base-bne')
#Examples of the tokenization performed by BERT·
example_text  =  ['Usando  BETO  en  clases  de  ciencia  de  datos  de  la  universidad
politécnica.', "Los estudiantes de este grado son muy aplicados y estudiosos."]
tokenized_text = tokenizer.tokenize(example_text[0])
print(tokenized_text)
tokenized_text = tokenizer.tokenize(example_text[1])
print(tokenized_text)
bert_input = tokenizer(example_text,padding='max_length', max_length = 20,
                      truncation=False, return_tensors="pt")
# Transform tokens to vocabulary indexes
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
#BERT WORKS WITH THESE REPRESENTATIONS OBTAINED FOR THE TOKENS
print(bert_input)
print(bert_input['input_ids'])
print(bert_input['attention_mask'])
#To transform the sequence of token indices to words in the texts
example_text = tokenizer.decode(bert_input.input_ids[0])
print(example_text)
example_text = tokenizer.decode(bert_input.input_ids[1])
print(example_text)
```

Obtaining the representation of the texts using pre-trained BETO

EXAMPLE 3

```
import torch
from transformers import BertTokenizer
tokenizer=BertTokenizer.from_pretrained('dccuchile/bert-base-spanish-wwm-uncased')
import pandas as pd
#The max_length parameter depends on the  texts' length in the dataset
#Path of the trainig datataset
df_huhu=pd.read_csv("path_trainig_dataset")
df_huhu.head(10)[["tweet","humor"]]
X, y=df_huhu["tweet"], df_huhu["humor"]

#Let  us  define  X  as  the  texts  from  the  HUHU  training  set.  We  use  the  100  examples
for demo purposes.
```

```
example_text=X[:100].to_list()
bert_input = tokenizer(example_text, padding='max_length', max_length = 30,
                      truncation=True, return_tensors="pt")
# Transform tokens to vocabulary indexes
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
#BERT WORKS WITH THESE REPRESENTATIONS OBTAINED FOR THE TOKENS
print(bert_input['input_ids'])
print(bert_input['token_type_ids'])
print(bert_input['attention_mask'])
model = BertModel.from_pretrained('dccuchile/bert-base-spanish-wwm-uncased')
#Setting the evaluation mode, this option does not make gradient updating
model.eval()
# Send the data to GPU
bert_input = bert_input.to('cuda')
model.to('cuda')
with torch.no_grad():
   outputs = model(**bert_input)
   # Transformers models always return tuples.
     #  Here,  the  first  element  corresponds  to  the  vectors  in  the  output  of  the
last BETO layer.
   encoded_layers = outputs[0]
   print(encoded_layers.size())
   #Here we obtain the embedding of the CLS tokens for each input text.
   #This representation serves as a contextual embedding of the texts.
   cls_vector = encoded_layers[:,0,:]
   print(cls_vector.size())
   #Vector associated with the CLS token of the first text in the entry.
   cls_vector = cls_vector.cpu().detach().numpy()[0]
   print(len(cls_vector))
```

Obtaining the representation of the texts using pre-trained RoBERTa

EXAMPLE 4

```
import torch
from transformers import RobertaTokenizer, RobertaModel
tokenizer = RobertaTokenizer.from_pretrained('PlanTL-GOB-ES/roberta-base-bne')
import pandas as pd
#The max_length parameter depends on the  texts' length in the dataset
#Path of the trainig datataset
df_huhu=pd.read_csv("/content/drive/MyDrive/LNR-2023/Boletines/PL3/train.csv")
df_huhu.head(10)[["tweet","humor"]]
X, y=df_huhu["tweet"], df_huhu["humor"]

#Let  us  define  X  as  the  texts  from  the  HUHU  training  set.  We  use  the  100  examples
for demo purposes.
```

```
example_text=X[:100].to_list()
print(example_text)
roberta_input = tokenizer(example_text, padding='max_length', max_length = 30,
                      truncation=True, return_tensors="pt")

# Transform tokens to vocabulary indexes
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
#BERT WORKS WITH THESE REPRESENTATIONS OBTAINED FOR THE TOKENS
print(roberta_input['input_ids'])
print(roberta_input['attention_mask'])
model = RobertaModel.from_pretrained('PlanTL-GOB-ES/roberta-base-bne')
#Setting the evaluation mode, this option does not make gradient updating
model.eval()
# Send the data to GPU
roberta_input = roberta_input.to('cuda')
model.to('cuda')
with torch.no_grad():
   outputs = model(**roberta_input)
   # Transformers models always return tuples.
    # Here, the first element corresponds to the vectors in the output of the last
BETO layer.
   encoded_layers = outputs[0]
   print(encoded_layers.size())
   #Here we obtain the embedding of the CLS tokens for each input text.
   #This representation serves as a contextual embedding of the texts.
   cls_vector = encoded_layers[:,0,:]
   print(cls_vector.size())
   #Vector associated with the CLS token of the first text in the entry.
   cls_vector = cls_vector.cpu().detach().numpy()[0]
   print(len(cls_vector))
```

Performing fine-tuning of BETO for classification task

As can be seen in the previous example, no training of the model is performed; only BETO is used to obtain the text embeddings. A more refined variant involves recalibrating the weights learned by the model based on training examples from the specific task to be solved. To do this, it is necessary to define:

- the Dataset class for data handling;
- the model itself, which includes BETO or RoBERTa + a classifier (MLP);
- the training loop;

- the validation loop.

After having these four elements, the model based on BETO or RoBERTa can be retrained, followed by making predictions on the test dataset.

Dataset torch class

EXAMPLE 5

```
import numpy as np
from torch import nn
class DatasetBinary(torch.utils.data.Dataset):
   def __init__(self, df, tokenizer, mode="train"):
       self.mode =mode
       if mode !="train":
           self.labels= np.asarray([0]*len( df['tweet']))
       else:
           self.labels = [int(label) for label in df['humor']]
             self.texts  =  [tokenizer(text,padding='max_length',  max_length  =  30,
truncation=True,return_tensors="pt") for text in df['tweet']]

   def classes(self):
       return self.labels

   def __len__(self):
       return len(self.labels)

   def get_batch_labels(self, idx):
       # Fetch a batch of labels
       return np.array(self.labels[idx])

   def get_batch_texts(self, idx):
       # Fetch a batch of inputs
       return self.texts[idx]

   def __getitem__(self, idx):
       batch_texts = self.get_batch_texts(idx)
       batch_y = self.get_batch_labels(idx)
       return batch_texts, batch_y
```

Training loop

EXAMPLE 6

```
from torch.optim import Adam
from tqdm import tqdm

#TRAINING LOOP
def    train(model,    train,    val,    learning_rate,    epochs,    batch_size=8,
loss=nn.CrossEntropyLoss()):
     train_dataloader  =  torch.utils.data.DataLoader(train,  batch_size=batch_size,
shuffle=True)
   val_dataloader = torch.utils.data.DataLoader(val, batch_size=batch_size)
   use_cuda = torch.cuda.is_available()
   device = torch.device("cuda" if use_cuda else "cpu")
   optimizer = Adam(model.parameters(), lr= learning_rate)
   if use_cuda:
           model = model.cuda()
           criterion = loss.cuda()
   for epoch_num in range(epochs):
           total_acc_train = 0
           total_loss_train = 0
           for train_input, train_label in tqdm(train_dataloader):
               train_label = train_label.to(device)
               mask = train_input['attention_mask'].to(device)
               input_id = train_input['input_ids'].squeeze(1).to(device)

               output = model(input_id, mask)
               batch_loss = loss(output, train_label)
               total_loss_train += batch_loss.item()
               acc = (output.argmax(dim=1) == train_label).sum().item()
               total_acc_train += acc
               model.zero_grad()
               batch_loss.backward()
               optimizer.step()
           #validation on the development set
           total_acc_val = 0
           total_loss_val = 0
           with torch.no_grad():
               for val_input, val_label in val_dataloader:
                   val_label = val_label.to(device)
                   mask = val_input['attention_mask'].to(device)
                   input_id = val_input['input_ids'].squeeze(1).to(device)
                   output = model(input_id, mask)
                   batch_loss = loss(output, val_label)
                   total_loss_val += batch_loss.item()
                   acc = (output.argmax(dim=1) == val_label).sum().item()
                   total_acc_val += acc
```

```
                print(f'Epochs: {epoch_num + 1} | Train Loss: {total_loss_train / len(train): .3f} \
                | Train Accuracy: {total_acc_train / len(train): .3f} \
                | Val Loss: {total_loss_val / len(val): .3f} \
                | Val Accuracy: {total_acc_val / len(val): .3f}')
```

Validation loop

EXAMPLE 7

```
def evaluate(model, test, batch_size=8, evaltype=True):
   test_dataloader = torch.utils.data.DataLoader(test, batch_size=batch_size)
   use_cuda = torch.cuda.is_available()
   device = torch.device("cuda" if use_cuda else "cpu")
   if use_cuda:
       model = model.cuda()
   total_acc_test = 0
   predict=[]
   out = None
   with torch.no_grad():
       k=0
       for test_input, test_label in test_dataloader:
             test_label = test_label.to(device)
             mask = test_input['attention_mask'].to(device)
             input_id = test_input['input_ids'].squeeze(1).to(device)
             output = model(input_id, mask)
             if k == 0:
                 out = output
             else:
                 out = torch.cat((out, output), 0)
             k+=1
             if evaltype:
               acc = (output.argmax(dim=1) == test_label).sum().item()
               total_acc_test += acc
   if evaltype:
       print(f'Test Accuracy: {total_acc_test / len(test_data): .3f}')
   return out.argmax(dim=1)
```

Fine-tuning BETO for binary classification

EXAMPLE 8

```
class BETOClassifier(nn.Module):
                                def          __init__(self,          dropout=0.3,
model_name='dccuchile/bert-base-spanish-wwm-uncased'):
       super(BETOClassifier, self).__init__()
       self.beto = BertModel.from_pretrained(model_name)
       self.dropout = nn.Dropout(dropout)
       self.linear = nn.Linear(768, 2)
       self.relu = nn.ReLU()

   def forward(self, input_id, mask):
                      _,   pooled_output   =   self.beto(input_ids   =   input_id,
attention_mask=mask,return_dict=False)
       dropout_output = self.dropout(pooled_output)
       linear_output = self.linear(dropout_output)
       final_layer = self.relu(linear_output)
       return final_layer


#Hyperparameters
EPOCHS = 10
BATCH = 16
LR = 1e-6
#Creating the model based on BETO
tokenizer                                                                     =
BertTokenizer.from_pretrained('dccuchile/bert-base-spanish-wwm-uncased')
traind = DatasetBinary(df_train, tokenizer)
val = DatasetBinary(df_val, tokenizer)
test = DatasetBinary(df_test, tokenizer, mode="test")
model = BETOClassifier()
#Fine-tuning the model
train(model, traind, val, LR, EPOCHS, BATCH)
#Validating the model in the test dataset
evaluate(model, test, BATCH, False)
```

Fine-tuning RobERTA for binary classification

EXAMPLE 9

```
class RoBERTaClassifier(nn.Module):
   def __init__(self, dropout=0.3, model_name='PlanTL-GOB-ES/roberta-base-bne'):
       super(RoBERTaClassifier, self).__init__()
       self.roberta = RobertaModel.from_pretrained(model_name)
       self.dropout = nn.Dropout(dropout)
```

```
        self.linear = nn.Linear(768, 2)
        self.relu = nn.ReLU()

    def forward(self, input_id, mask):
                    _,    pooled_output    =    self.roberta(input_ids    =    input_id,
attention_mask=mask,return_dict=False)
        dropout_output = self.dropout(pooled_output)
        linear_output = self.linear(dropout_output)
        final_layer = self.relu(linear_output)
        return final_layer


#Fine-tuninig and validating the RoBERTa-based model for binary classification.
#Hyperparameters
EPOCHS = 10
BATCH = 16
LR = 1e-6
#Creating the model based on BETO
tokenizer = RobertaTokenizer.from_pretrained('PlanTL-GOB-ES/roberta-base-bne')
traind = DatasetBinary(df_train, tokenizer)
val = DatasetBinary(df_val, tokenizer)
test = DatasetBinary(df_test, tokenizer, mode="test")
model = RoBERTaClassifier()
#Fine-tuning the model
train(model, traind, val, LR, EPOCHS, BATCH)
#Validating the model in the test dataset
evaluate(model, test, BATCH, False)
```

To enable multi-label classification, several adjustments are required. Firstly, the Dataset class should be updated (DatasetMulti) to incorporate the relevant labels for HUHU task 2a. Secondly, modifications are needed for the loss function and the output of the BETOClassifier (BETOClassifierMulti) and RoBERTaClassifier (RoBERTaClassifierMulti). In particular, **the BCELoss** should be utilized during the training loop, and the **sigmoid activation** function needs to be applied to the output layer

Modified Dataset class for multi-label task

EXAMPLE 10

```
import numpy as np
from torch import nn
class DatasetMulti(torch.utils.data.Dataset):
   def __init__(self, df, tokenizer, mode="train"):
       self.mode =mode
       if mode !="train":
           self.labels= np.asarray([0]*len(df['tweet']))
       else:
                                   self.labels   =   [[int(labels["prejudice_woman"]),
int(labels["prejudice_lgbtiq"]),
int(labels["prejudice_inmigrant_race"]),int(labels["gordofobia"])]   for   _,
labels              in              df[["prejudice_woman","prejudice_lgbtiq",
"prejudice_inmigrant_race", "gordofobia"]].iterrows()]
         self.texts = [tokenizer(text,padding='max_length', max_length = 30,
truncation=True,return_tensors="pt") for text in df['tweet']]
       print("DATASET info",len(self.labels), len(self.texts))

   def classes(self):
       return self.labels

   def __len__(self):
       return len(self.labels)

   def get_batch_labels(self, idx):
       # Fetch a batch of labels
       return np.array(self.labels[idx])

   def get_batch_texts(self, idx):
       # Fetch a batch of inputs
       return self.texts[idx]

   def __getitem__(self, idx):
       batch_texts = self.get_batch_texts(idx)
       batch_y = self.get_batch_labels(idx)
       return batch_texts, batch_y
```

Changes for fine-tuning BETO for multi-label classification

EXAMPLE 11

```
class BETOClassifierMulti(nn.Module):
                       def   __init__(self,   dropout=0.3,   labels=4,
model_name='dccuchile/bert-base-spanish-wwm-uncased'):
       super(BETOClassifierMulti, self).__init__()
       self.beto = BertModel.from_pretrained(model_name)
       self.dropout = nn.Dropout(dropout)
       self.linear = nn.Linear(768, labels)
       self.sigm = nn.Sigmoid()

   def forward(self, input_id, mask):
                       _,  pooled_output  =  self.beto(input_ids  =  input_id,
attention_mask=mask,return_dict=False)
       dropout_output = self.dropout(pooled_output)
       linear_output = self.linear(dropout_output)
       final_layer = self.sigm(linear_output)
       return final_layer

#Hyperparameters
EPOCHS = 10
BATCH = 16
LR = 1e-6
#Creating the model based on BETO
tokenizer                                                                   =
BertTokenizer.from_pretrained('dccuchile/bert-base-spanish-wwm-uncased')
traind = DatasetMulti(df_train, tokenizer)
val = DatasetMulti(df_val, tokenizer)
test = DatasetMulti(df_test, tokenizer, mode="test")
model = BETOClassifierMulti()
#Fine-tuning the model
train(model, traind, val, LR, EPOCHS, BATCH, loss=nn.BCELoss())
#Validating the model in the test dataset
evaluate(model, test, BATCH, False)
```

Changes for fine-tuning RoBERTA for multi-label classification

EXAMPLE 12

```
from torch import nn
class RoBERTaClassifierMulti(nn.Module):
                            def     __init__(self,     dropout=0.3,labels=4,
model_name='PlanTL-GOB-ES/roberta-base-bne'):
       super(RoBERTaClassifierMulti, self).__init__()
       self.roberta = RobertaModel.from_pretrained(model_name)
```

```
        self.dropout = nn.Dropout(dropout)
        self.linear = nn.Linear(768,labels)
        self.sigm = nn.Sigmoid()

    def forward(self, input_id, mask):
                        _,    pooled_output    =    self.roberta(input_ids    =    input_id,
attention_mask=mask,return_dict=False)
        dropout_output = self.dropout(pooled_output)
        linear_output = self.linear(dropout_output)
        final_layer = self.sigm(linear_output)
        return final_layer


##Fine-tuninig and validating the RoBERTa-based model for binary classification.
#Hyperparameters
EPOCHS = 10
BATCH = 32
LR = 1e-6
tokenizer = RobertaTokenizer.from_pretrained('PlanTL-GOB-ES/roberta-base-bne')
traind = DatasetMulti(df_train, tokenizer)
val = DatasetMulti(df_val, tokenizer)
test = DatasetMulti(df_test, tokenizer, mode="test")
#Creating the model based on BETO
model = RoBERTaClassifierMulti()
#Fine-tuning the model
train(model, traind, val, LR, EPOCHS, BATCH, loss=nn.BCELoss())
#Validating the model in the test dataset
evaluate(model, test, BATCH, False)
```

To perform regression, several modifications are required. Firstly, the Dataset class should be updated (DatasetRegression) to include the appropriate dependent variable for task 2b of HUHU. Secondly, changes are needed for the loss function and the output of the BETOClassifier (BETORegression) and RoBERTaClassifier (RoBERTaRegression). Specifically, the MSELoss function should be utilized during the training loop, and the Linear activation function should be applied to the output layer.

Modified Dataset class for regression task

EXAMPLE 13

```
import numpy as np
from torch import nn
class DatasetRegression(torch.utils.data.Dataset):
   def __init__(self, df, tokenizer, mode="train"):
       self.mode =mode
       if mode !="train":
           self.labels= np.asarray([0]*len( df['tweet']))
       else:
           self.labels = [int(label) for label in df['mean_prejudice']]
             self.texts  =  [tokenizer(text,padding='max_length',  max_length  =  30,
truncation=True,return_tensors="pt") for text in df['tweet']]

   def classes(self):
       return self.labels

   def __len__(self):
       return len(self.labels)

   def get_batch_labels(self, idx):
       # Fetch a batch of labels
       return np.array(self.labels[idx])

   def get_batch_texts(self, idx):
       # Fetch a batch of inputs
       return self.texts[idx]

   def __getitem__(self, idx):
       batch_texts = self.get_batch_texts(idx)
       batch_y = self.get_batch_labels(idx)
       return batch_texts, batch_y
```

Changes for fine-tuning BETO for regression

EXAMPLE 14

```
class BETORegression(nn.Module):
                                 def         __init__(self,           dropout=0.3,
model_name='dccuchile/bert-base-spanish-wwm-uncased'):
       super(BETORegression, self).__init__()
       self.beto = BertModel.from_pretrained(model_name)
       self.dropout = nn.Dropout(dropout)
       self.linear = nn.Linear(768, 1)
```

```
   def forward(self, input_id, mask):
                       _,   pooled_output   =   self.beto(input_ids   =   input_id,
attention_mask=mask,return_dict=False)
       dropout_output = self.dropout(pooled_output)
       final_layer = self.linear(dropout_output)
       return final_layer

#Hyperparameters
EPOCHS = 1
BATCH = 16
LR = 1e-6
#Creating the regression model based on BETO
tokenizer                                                                             =
BertTokenizer.from_pretrained('dccuchile/bert-base-spanish-wwm-uncased')
traind = DatasetRegression(df_train,tokenizer)
val= DatasetRegression(df_val, tokenizer)
test= DatasetRegression(df_test, tokenizer, mode="test")

model = BETORegression()
#Fine-tuning the model
train(model, traind, val, LR, EPOCHS, BATCH,  loss=nn.MSELoss())
#Validating the model in the test dataset
evaluate(model, test, BATCH, False)
```

Changes for fine-tuning RoBERTa for regression

EXAMPLE 15

```
class RoBERTaRegression(nn.Module):
   def __init__(self, dropout=0.3, model_name='PlanTL-GOB-ES/roberta-base-bne'):
       super(RoBERTaRegression, self).__init__()
       self.roberta = RobertaModel.from_pretrained(model_name)
       self.dropout = nn.Dropout(dropout)
       self.linear = nn.Linear(768,1)

   def forward(self, input_id, mask):
                     _,   pooled_output   =   self.roberta(input_ids   =   input_id,
attention_mask=mask,return_dict=False)
       dropout_output = self.dropout(pooled_output)
       final_layer = self.linear(dropout_output)
       return final_layer

  #Hyperparameters
EPOCHS = 10
BATCH = 16
```

```
LR = 1e-6
#Creating the regression model based on RoBERTa
tokenizer = RobertaTokenizer.from_pretrained('PlanTL-GOB-ES/roberta-base-bne')
traind = DatasetRegression(df_train,tokenizer)
val= DatasetRegression(df_val, tokenizer)
test= DatasetRegression(df_test, tokenizer, mode="test")

model = RoBERTaRegression()
#Fine-tuning the model
train(model, traind, val, LR, EPOCHS, BATCH, loss=nn.MSELoss())
#Validating the model in the test dataset
evaluate(model, test, BATCH, False)
```

ABLATION ANALYSIS

The aim of this section is to introduce error analysis as a procedure to identify potential causes for classifier failures in specific predictions.
Summary of steps for creating a text classification system:

1. Set up the training/validation dataset and select an evaluation metric to assess the classifier's effectiveness and/or efficiency (e.g., average F1 score).
2. Create an initial model:
   a. Train the model using the training dataset.
   b. Evaluate and adjust parameters using the validation dataset. If the data is not pre-partitioned, employ a validation scheme (e.g., K-Fold cross-validation).

How to identify and address errors?

- Analyze potential causes of errors to rectify them.

General concept: Iteratively identify errors to correct them and enhance the classifier's effectiveness. Manual examination of the model's errors can provide valuable insights for determining the next steps.
For example, let's suppose we have a binary classifier C1, employed for text classification, and it exhibits a 40% validation error rate. Further analysis reveals that C1 is incorrectly assigning texts from *"class 0"*

(prejudice) to *"class 1"* (prejudiced humor). In such a scenario, should we focus on improving C1's performance specifically for *"class 0"*?

Rather than investing significant time in evaluating various parameter combinations or altering text representations, which may not yield significant benefits in the end, error analysis allows us to precisely determine whether such efforts are likely to produce the desired outcomes.

GUIDELINE

1. Sampling a subset of errors made by the classifier from the validation data is crucial, typically around ~100 examples if the validation set is not extensive or 20% of errors in large validation sets.
2. It is essential to quantify the number of errors specifically pertaining to the class of interest, such as *"class 0."* If only 5% of texts in this class, denoted as *"class 1*," are misclassified, and we manage to completely rectify this problem, the error rate will decrease at most to 35% from the initial 40%. However, if a significant 30% of texts from *"class 1"* are misclassified, there is a greater potential for improving the model's efficacy, with the hope of substantially reducing the error rate.

It is beneficial to create a table based on the identified errors and their representativeness for analyzing potential error types. This approach enables the implementation of new strategies to address these errors, ultimately selecting the variant that yields the most significant performance improvement.

TEXT IDs	**Class 0**	**Class 1**	**Text with URLs**	**Very large texts**	**No embedding**	**Misspelling**	**Idioms**	**Figurative language**
1	X		X			X		
2		X		X		X		
3		X						
4	X				X	X		
...			X				X	
...					X			
...			X					
99		X				X		X
% Errors	**30%**	**70%**	**15%**	**5%**	**20%**	**50%**	**5%**	**5%**

The conclusion of this process gives us an estimation of the value of working on each of these different error categories and where to prioritize our efforts. In the previous table, it is clear that a significant number of errors occur in texts with spelling issues, followed by those where obtaining their embedding representation is not possible. It is also demonstrated that no matter how much we concentrate on *"class 0"*, the maximum reduction in errors will be 30% at best. Hence, it would be beneficial to focus on correcting spelling errors and enhancing the embedding-based representation.

ACTIVITY

Build classification models for HUHU tasks 1, 2a, and 2b using BETO and RoBERTa, two pre-trained models specifically designed for the Spanish language. These models can extract valuable representations (feature engines) or be fine-tuned on the HUHU dataset. One approach involves utilizing the token embedding of [CLS] as the text representation and training machine learning models. Another option is to perform fine-tuning on the HUHU data using a classifier based on BETOor RoBERTa for Spanish. Deliver code and report with explanation of the strategy used.

PARTICIPATION IN DETESTS AND REPORT

Regarding the submission of the HUHU task results, please note the following:

The deadline for sending the runs to the task organizers is Saturday, May 20th, 2023. Teams of 3 people should submit 5 runs, while teams of 2 people should submit 3 runs, one for each subtask. Individual participation is also allowed. In addition, a report (per team) of 4/5 pages must be submitted, providing a brief description of the work performed. The report should include the following details:

Name(s)
Team name registered for the evaluation task (HUHU)
Preprocessing techniques (if you applied any techniques)
Text representation approach
Model(s) used and any model combination employed
Utilization of BETO or RoBERTa and its role, if applicable
Validation strategy adopted
Parameter adjustment strategy employed
Results obtained (with discussion of the evaluation metric used)
Error analysis conducted
The report must be submitted on PoliformaT by 1st June, 2023

The Google collaborative environment (Google Colab) can be used to implement the machine learning models:
https://colab.research.google.com/notebooks/welcome.ipynb#scrollTo=5fCEDCU_qrC0