```python
#!pip install langdetect

#!pip install contractions

#!pip install imblearn

# Libraries for general purpose
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Text cleaning
import re
import string
import emoji
import nltk
from nltk.stem import WordNetLemmatizer, PorterStemmer
from nltk.corpus import stopwords

# Data preprocessing
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import RandomOverSampler
from langdetect import detect, LangDetectException
import contractions
from nltk.tokenize import word_tokenize

# Naive Bayes
# from sklearn.feature_extraction.text import CountVectorizer
# from sklearn.feature_extraction.text import TfidfTransformer
# from sklearn.naive_bayes import MultinomialNB

# PyTorch LSTM
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader, RandomSampler,
SequentialSampler

# # Tokenization for LSTM
# from collections import Counter
# from gensim.models import Word2Vec

# Transformers library for BERT
import transformers
from transformers import BertModel
from transformers import BertTokenizer
from transformers import AdamW, get_linear_schedule_with_warmup
from sklearn.metrics import classification_report, confusion_matrix
```

```
import time

# Set seed for reproducibility
import random
seed_value = 2042
random.seed(seed_value)
np.random.seed(seed_value)
torch.manual_seed(seed_value)
torch.cuda.manual_seed_all(seed_value)

# Set style for plots
sns.set_style("whitegrid")
sns.despine()
plt.style.use("seaborn-whitegrid")
plt.rc("figure", autolayout=True)
plt.rc("axes", labelweight="bold", labelsize="large",
titleweight="bold", titlepad=10)

# Define stop words for text cleaning
stop_words = set(stopwords.words('english'))

# Initialize lemmatizer for text cleaning
lemmatizer = WordNetLemmatizer()

import os

df = pd.read_csv("cyberbullying_tweets.csv")
#df = pd.read_csv(os.path.join(r'jigsaw-toxic-comment-classification-
challenge', 'cyberbullying_tweets.csv'), encoding = 'ISO-8859-1')
df.head()

#rename columns
df = df.rename(columns={'tweet_text': 'text', 'cyberbullying_type':
'sentiment'})

#check duplicated tweets
df.duplicated().sum()

36

#remove duplicated tweets
df = df[~df.duplicated()]

#check if classes are balanced
df.sentiment.value_counts()
```

Text deep cleaning

```
!pip install demoji
```

```python
import demoji
# Clean emojis from text
def strip_emoji(text):
    # old version -> return emoji.get_emoji_regexp().sub("", text)
    return demoji.replace(text, '')

# Remove punctuations, stopwords, links, mentions and new line
characters
def strip_all_entities(text):
    text = re.sub(r'\r|\n', ' ', text.lower())  # Replace newline and
carriage return with space, and convert to lowercase
    text = re.sub(r"(?:\@|https?\://)\S+", "", text)  # Remove links
and mentions
    text = re.sub(r'[^\x00-\x7f]', '', text)  # Remove non-ASCII
characters
    banned_list = string.punctuation
    table = str.maketrans('', '', banned_list)
    text = text.translate(table)
    text = ' '.join(word for word in text.split() if word not in
stop_words)
    return text

# Clean hashtags at the end of the sentence, and keep those in the
middle of the sentence by removing just the # symbol
def clean_hashtags(tweet):
    # Remove hashtags at the end of the sentence
    new_tweet = re.sub(r'(\s+#[\w-]+)+\s*$', '', tweet).strip()

    # Remove the # symbol from hashtags in the middle of the sentence
    new_tweet = re.sub(r'#([\w-]+)', r'\1', new_tweet).strip()

    return new_tweet

# Filter special characters such as & and $ present in some words
def filter_chars(text):
    return ' '.join('' if ('$' in word) or ('&' in word) else word for
word in text.split())

# Remove multiple spaces
def remove_mult_spaces(text):
    return re.sub(r"\s\s+", " ", text)

# Function to check if the text is in English, and return an empty
string if it's not
def filter_non_english(text):
    try:
        lang = detect(text)
    except LangDetectException:
        lang = "unknown"
    return text if lang == "en" else ""
```

```python
# Expand contractions
def expand_contractions(text):
    return contractions.fix(text)

# Remove numbers
def remove_numbers(text):
    return re.sub(r'\d+', '', text)

# Lemmatize words
def lemmatize(text):
    words = word_tokenize(text)
    lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
    return ' '.join(lemmatized_words)

# Remove short words
def remove_short_words(text, min_len=2):
    words = text.split()
    long_words = [word for word in words if len(word) >= min_len]
    return ' '.join(long_words)

# Replace elongated words with their base form
def replace_elongated_words(text):
    regex_pattern = r'\b(\w+)((\w)\3{2,})(\w*)\b'
    return re.sub(regex_pattern, r'\1\3\4', text)

# Remove repeated punctuation
def remove_repeated_punctuation(text):
    return re.sub(r'[\?\.\!]+(?=[\?\.\!])', '', text)

# Remove extra whitespace
def remove_extra_whitespace(text):
    return ' '.join(text.split())

def remove_url_shorteners(text):
    return re.sub(r'(?:http[s]?://)?(?:www\.)?(?:bit\.ly|goo\.gl|
t\.co|tinyurl\.com|tr\.im|is\.gd|cli\.gs|u\.nu|url\.ie|tiny\.cc|
alturl\.com|ow\.ly|bit\.do|adoro\.to)\S+', '', text)

# Remove spaces at the beginning and end of the tweet
def remove_spaces_tweets(tweet):
    return tweet.strip()

# Remove short tweets
def remove_short_tweets(tweet, min_words=3):
    words = tweet.split()
    return tweet if len(words) >= min_words else ""

# Function to call all the cleaning functions in the correct order
def clean_tweet(tweet):
    tweet = strip_emoji(tweet)
```

```
    tweet = expand_contractions(tweet)
    tweet = filter_non_english(tweet)
    tweet = strip_all_entities(tweet)
    tweet = clean_hashtags(tweet)
    tweet = filter_chars(tweet)
    tweet = remove_mult_spaces(tweet)
    tweet = remove_numbers(tweet)
    tweet = lemmatize(tweet)
    tweet = remove_short_words(tweet)
    tweet = replace_elongated_words(tweet)
    tweet = remove_repeated_punctuation(tweet)
    tweet = remove_extra_whitespace(tweet)
    tweet = remove_url_shorteners(tweet)
    tweet = remove_spaces_tweets(tweet)
    tweet = remove_short_tweets(tweet)
    tweet = ' '.join(tweet.split())  # Remove multiple spaces between
words
    return tweet

df['text_clean'] = [clean_tweet(tweet) for tweet in df['text']]

df.head()
```

Check duplicated tweets after clean and remove them

```
print(f'There are around {int(df["text_clean"].duplicated().sum())}
duplicated tweets, we will remove them.')
df.drop_duplicates("text_clean", inplace=True)
df.sentiment.value_counts()

''' Since the class is very unbalanced compared to the other classes
and looks too "generic",
    we decide to remove the tweets labeled belonging to this class.
    EDIT: by performing some tests, the f1 score for predicting the
"other_cyberbullying" resulted to be around 60%,
    a value far lower compared to the othter f1 scores (around 95%
using LSTM model).
    This supports the decision of removing this generic class.
'''
df = df[df["sentiment"]!="other_cyberbullying"]

sentiments = ["religion","age","ethnicity","gender","not bullying"]
#list of the classes names, which will be useful for the future plots.
```

Tweet length analyze

```
df['text_len'] = [len(text.split()) for text in df.text_clean]

plt.figure(figsize=(7,5))
ax = sns.countplot(x='text_len', data=df[df['text_len']<10],
```

```
palette='mako')
plt.title('Count of tweets with less than 10 words', fontsize=20)
plt.yticks([])
ax.bar_label(ax.containers[0])
plt.ylabel('count')
plt.xlabel('')
plt.show()
```

Check tweets' length

```
df.sort_values(by=['text_len'], ascending=False)

plt.figure(figsize=(16,5))
ax = sns.countplot(x='text_len', data=df[(df['text_len']<=1000) &
(df['text_len']>10)], palette='Blues_r')
plt.title('Count of tweets with high number of words', fontsize=25)
plt.yticks([])
ax.bar_label(ax.containers[0])
plt.ylabel('count')
plt.xlabel('')
plt.show()

#remove tweets that are long than 100 words
df = df[df['text_len'] < df['text_len'].quantile(0.995)]

max_len = np.max(df['text_len']) #get a length of the longest tweets
max_len

31

df.sort_values(by=["text_len"], ascending=False)

#sentiment baganii utguudiig kodloh
df['sentiment'] =
df['sentiment'].replace({'religion':0,'age':1,'ethnicity':2,'gender':3
,'not_cyberbullying':4})
```

BERT classification

```
#splitting dataset for train and test
X = df['text_clean'].values
y = df['sentiment'].values
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, stratify=y, random_state=seed_value)

#splitting dataset for train and validation
X_train, X_valid, y_train, y_valid = train_test_split(X_train,
y_train, test_size=0.2, stratify=y_train, random_state=seed_value)
```

```
(unique, counts) = np.unique(y_train, return_counts=True)
np.asarray((unique, counts)).T
```

Oversampling of training set

```
#The classes are unbalanced, so need to oversample the training set
such that all classes have the same count as the most populated one.
ros = RandomOverSampler()
X_train_os, y_train_os = ros.fit_resample(np.array(X_train).reshape(-
1,1),np.array(y_train).reshape(-1,1))

X_train_os = X_train_os.flatten()
y_train_os = y_train_os.flatten()

(unique, counts) = np.unique(y_train, return_counts=True)
np.asarray((unique, counts)).T
```

BERT tokenization

```
''' Since we need to tokenize the tweets (get "input ids" and
"attention masks") for BERT,
    we load the specific BERT tokenizer from the Hugging Face library.
'''

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',
do_lower_case=True)

def bert_tokenizer(data):
    input_ids = []
    attention_masks = []
    for sent in data:
        encoded_sent = tokenizer.encode_plus(
            text=sent,
            add_special_tokens=True,        # Add `[CLS]` and `[SEP]`
special tokens
            max_length=MAX_LEN,             # Choose max length to
truncate/pad
            pad_to_max_length=True,         # Pad sentence to max
length
            return_attention_mask=True      # Return attention mask
            )
        input_ids.append(encoded_sent.get('input_ids'))
        attention_masks.append(encoded_sent.get('attention_mask'))

    # Convert lists to tensors
    input_ids = torch.tensor(input_ids)
    attention_masks = torch.tensor(attention_masks)

    return input_ids, attention_masks
```

```
# ''' Since we need to specify the length of the longest tokenized
sentence,
#     we tokenize the train tweets using the "encode" method of the
original
#     BERT tokenizer and check the longest sentence.
# '''
# # Tokenize train tweets
# encoded_tweets = [tokenizer.encode(sent, add_special_tokens=True)
for sent in X_train]

# # Find the longest tokenized tweet
# max_len = max([len(sent) for sent in encoded_tweets])
# print('Max length: ', max_len)  ->>> not working!!!!

# Max length is 82. So we can choose the max length as 128.
MAX_LEN = 128

train_inputs, train_masks = bert_tokenizer(X_train_os)
val_inputs, val_masks = bert_tokenizer(X_valid)
test_inputs, test_masks = bert_tokenizer(X_test)
```

Data preprocessing for PyTorch BERT model

```
''' Since we are using the BERT model built on PyTorch,
    we need to convert the arrays to pytorch tensors and
    create dataloaders for the data.
'''
# Convert target columns to pytorch tensors format
train_labels = torch.from_numpy(y_train_os)
val_labels = torch.from_numpy(y_valid)
test_labels = torch.from_numpy(y_test)
```

Dataloader

```
# To fine-tune the BERT model, the original authors recommend a batch
size of 16 or 32.
batch_size = 32

# Create the DataLoader for our training set
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler,
batch_size=batch_size)

# Create the DataLoader for our validation set
val_data = TensorDataset(val_inputs, val_masks, val_labels)
val_sampler = SequentialSampler(val_data)
val_dataloader = DataLoader(val_data, sampler=val_sampler,
batch_size=batch_size)
```

```
# Create the DataLoader for our test set
test_data = TensorDataset(test_inputs, test_masks, test_labels)
test_sampler = SequentialSampler(test_data)
test_dataloader = DataLoader(test_data, sampler=test_sampler,
batch_size=batch_size)
```

BERT Modeling

```
'''
    Now we can create a custom BERT classifier class, including the
original BERT model
    (made of transformer layers) and additional Dense layers to
perform the desired classification task.
'''
class Bert_Classifier(nn.Module):
    def __init__(self, freeze_bert=False):
        super(Bert_Classifier, self).__init__()
        # Specify hidden size of BERT, hidden size of the classifier,
and number of labels
        n_input = 768
        n_hidden = 50
        n_output = 5

        # Instantiate BERT model
        self.bert = BertModel.from_pretrained('bert-base-uncased')

        # Instantiate the classifier (a fully connected layer followed
by a ReLU activation and another fully connected layer)
        self.classifier = nn.Sequential(
            nn.Linear(n_input, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_output)
        )

        # Freeze the BERT model weights if freeze_bert is True (useful
for feature extraction without fine-tuning)
        if freeze_bert:
            for param in self.bert.parameters():
                param.requires_grad = False

    def forward(self, input_ids, attention_mask):
        # Feed input data (input_ids and attention_mask) to BERT
        outputs = self.bert(input_ids=input_ids,
                            attention_mask=attention_mask)

        # Extract the last hidden state of the `[CLS]` token from the
BERT output (useful for classification tasks)
        last_hidden_state_cls = outputs[0][:, 0, :]

        # Feed the extracted hidden state to the classifier to compute
```

```
logits
        logits = self.classifier(last_hidden_state_cls)

        return logits

# Function for initializing the BERT Classifier model, optimizer, and
learning rate scheduler
def initialize_model(epochs=4):
    # Instantiate Bert Classifier
    bert_classifier = Bert_Classifier(freeze_bert=False)

    bert_classifier.to(device)

    # Set up optimizer
    optimizer = AdamW(bert_classifier.parameters(),
                      lr=5e-5,    # learning rate, set to default
value
                      eps=1e-8    # decay, set to default value
                      )

    # Calculate total number of training steps
    total_steps = len(train_dataloader) * epochs

    # Define the learning rate scheduler
    scheduler = get_linear_schedule_with_warmup(optimizer,
                                                num_warmup_steps=0, #
Default value

num_training_steps=total_steps)
    return bert_classifier, optimizer, scheduler

# specify the use of GPU if present (highly recommend for the fine
tune)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
EPOCHS=2

# intialize the BERT model calling the "initialize_model" function we
defined.
bert_classifier, optimizer, scheduler =
initialize_model(epochs=EPOCHS)
```

BERT training

```
# Define Cross entropy Loss function for the multiclass classification
task
loss_fn = nn.CrossEntropyLoss()

def bert_train(model, train_dataloader, val_dataloader=None, epochs=4,
evaluation=False):
```

```python
    print("Start training...\n")
    for epoch_i in range(epochs):
        print("-"*10)
        print("Epoch : {}".format(epoch_i+1))
        print("-"*10)
        print("-"*38)
        print(f"{'BATCH NO.':^7} | {'TRAIN LOSS':^12} | {'ELAPSED
(s)':^9}")
        print("-"*38)

        # Measure the elapsed time of each epoch
        t0_epoch, t0_batch = time.time(), time.time()

        # Reset tracking variables at the beginning of each epoch
        total_loss, batch_loss, batch_counts = 0, 0, 0

        ###TRAINING###

        # Put the model into the training mode
        model.train()

        for step, batch in enumerate(train_dataloader):
            batch_counts +=1

            b_input_ids, b_attn_mask, b_labels = tuple(t.to(device)
for t in batch)

            # Zero out any previously calculated gradients
            model.zero_grad()

            # Perform a forward pass and get logits.
            logits = model(b_input_ids, b_attn_mask)

            # Compute loss and accumulate the loss values
            loss = loss_fn(logits, b_labels)
            batch_loss += loss.item()
            total_loss += loss.item()

            # Perform a backward pass to calculate gradients
            loss.backward()

            # Clip the norm of the gradients to 1.0 to prevent
"exploding gradients"
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

            # Update model parameters:
            # fine tune BERT params and train additional dense layers
            optimizer.step()
            # update learning rate
            scheduler.step()
```

```
            # Print the loss values and time elapsed for every 100
batches
            if (step % 100 == 0 and step != 0) or (step ==
len(train_dataloader) - 1):
                # Calculate time elapsed for 20 batches
                time_elapsed = time.time() - t0_batch

                print(f"{step:^9} | {batch_loss / batch_counts:^12.6f}
| {time_elapsed:^9.2f}")

                # Reset batch tracking variables
                batch_loss, batch_counts = 0, 0
                t0_batch = time.time()

        # Calculate the average loss over the entire training data
        avg_train_loss = total_loss / len(train_dataloader)

        ###EVALUATION###

        # Put the model into the evaluation mode
        model.eval()

        # Define empty lists to host accuracy and validation for each
batch
        val_accuracy = []
        val_loss = []

        for batch in val_dataloader:
            batch_input_ids, batch_attention_mask, batch_labels =
tuple(t.to(device) for t in batch)

            # We do not want to update the params during the
evaluation,
            # So we specify that we dont want to compute the gradients
of the tensors
            # by calling the torch.no_grad() method
            with torch.no_grad():
                logits = model(batch_input_ids, batch_attention_mask)

            loss = loss_fn(logits, batch_labels)

            val_loss.append(loss.item())

            # Get the predictions starting from the logits (get index
of highest logit)
            preds = torch.argmax(logits, dim=1).flatten()

            # Calculate the validation accuracy
            accuracy = (preds == batch_labels).cpu().numpy().mean() *
```

```
100
            val_accuracy.append(accuracy)

        # Compute the average accuracy and loss over the validation
set
        val_loss = np.mean(val_loss)
        val_accuracy = np.mean(val_accuracy)

        # Print performance over the entire training data
        time_elapsed = time.time() - t0_epoch
        print("-"*61)
        print(f"{'AVG TRAIN LOSS':^12} | {'VAL LOSS':^10} | {'VAL
ACCURACY (%)':^9} | {'ELAPSED (s)':^9}")
        print("-"*61)
        print(f"{avg_train_loss:^14.6f} | {val_loss:^10.6f} |
{val_accuracy:^17.2f} | {time_elapsed:^9.2f}")
        print("-"*61)
        print("\n")

    print("Training complete!")

bert_train(bert_classifier, train_dataloader, val_dataloader,
epochs=EPOCHS)
```

BERT prediction

```
'''Now we define a function similar to the model "evaluation", where
we feed to the model the test data instead of the validation data.'''
def bert_predict(model, test_dataloader):

    # Define empty list to host the predictions
    preds_list = []

    # Put the model into evaluation mode
    model.eval()

    for batch in test_dataloader:
        batch_input_ids, batch_attention_mask = tuple(t.to(device) for
t in batch)[:2]

        # Avoid gradient calculation of tensors by using "no_grad()"
method
        with torch.no_grad():
            logit = model(batch_input_ids, batch_attention_mask)

        # Get index of highest logit
        pred = torch.argmax(logit,dim=1).cpu().numpy()
        # Append predicted class to list
        preds_list.extend(pred)
```

```
    return preds_list

# call the defined function and get the class predictions of the test
data
bert_preds = bert_predict(bert_classifier, test_dataloader)

print('Classification Report for BERT :\n',
classification_report(y_test, bert_preds, target_names=sentiments))

def conf_matrix(y, y_pred, title, labels):
    fig, ax =plt.subplots(figsize=(7.5,7.5))
    ax=sns.heatmap(confusion_matrix(y, y_pred), annot=True,
cmap="Purples", fmt='g', cbar=False, annot_kws={"size":30})
    plt.title(title, fontsize=25)
    ax.xaxis.set_ticklabels(labels, fontsize=16)
    ax.yaxis.set_ticklabels(labels, fontsize=14.5)
    ax.set_ylabel('Test', fontsize=25)
    ax.set_xlabel('Predicted', fontsize=25)
    plt.show()

conf_matrix(y_test, bert_preds,' BERT Sentiment Analysis\nConfusion
Matrix', sentiments)
```