allenging-experiment-malayalam2-1

April 22, 2024

0.0.1 Performing checks for the resources available

[1]:
```
import torch

# If there's a GPU available...
if torch.cuda.is_available():

    # Tell PyTorch to use the GPU.
    device = torch.device("cuda")

    print('There are %d GPU(s) available.' % torch.cuda.device_count())

    print('We will use the GPU:', torch.cuda.get_device_name(0))

# If not...
else:
    print('No GPU available, using the CPU instead.')
    device = torch.device("cpu")
```

```
There are 1 GPU(s) available.
We will use the GPU: Tesla T4
```

[2]:
```
!nvidia-smi
```

```
Mon Apr 22 07:48:11 2024
+---------------------------------------------------------------------------------
--------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version:
12.2     |
|-----------------------------------------+----------------------+--------------
--------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
MIG M. |
|=========================================+======================+==============
```

```
========|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
0 |
| N/A   49C    P8              12W /  70W |      3MiB / 15360MiB |      0%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+-------------
--------+

+------------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                           GPU
Memory |
|        ID   ID
Usage      |
|==============================================================================
========|
|  No running processes found
|
+------------------------------------------------------------------------------
--------+
```

0.0.2 Installing huggingface transformers library

[3]:
```
!pip install transformers
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-
packages (4.38.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from transformers) (3.13.4)
Requirement already satisfied: huggingface-hub<1.0,>=0.19.3 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.20.3)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-
packages (from transformers) (1.25.2)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from transformers) (24.0)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-
packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (2023.12.25)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from transformers) (2.31.0)
Requirement already satisfied: tokenizers<0.19,>=0.14 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.15.2)
Requirement already satisfied: safetensors>=0.4.1 in
```

```
/usr/local/lib/python3.10/dist-packages (from transformers) (0.4.3)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-
packages (from transformers) (4.66.2)
Requirement already satisfied: fsspec>=2023.5.0 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.19.3->transformers) (2023.6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.19.3->transformers) (4.11.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->transformers) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (2024.2.2)
```

0.1 Importing necessary packages

[4]:
```python
import pandas as pd
import numpy as np
import random
from sklearn.preprocessing import LabelEncoder
from sklearn.utils.class_weight import compute_class_weight
import time
import datetime
from sklearn.metrics import classification_report,confusion_matrix
import random
import time
import torch.nn as nn
from transformers import AutoModel , AutoModelForSequenceClassification,␣
↪AutoConfig , AutoTokenizer , AdamW ,get_linear_schedule_with_warmup
import torch
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import rc
from matplotlib.ticker import MaxNLocator
```

0.2 Setting some configuration values

[5]:
```python
# Use plot styling from seaborn.
sns.set(style='darkgrid')

# Increase the plot size and font size.
sns.set(font_scale=1.5)
plt.rcParams["figure.figsize"] = (16,12)
```

```
# Set the seed value all over the place to make this reproducible.
seed_val = 42

random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)
```

0.3 Importing drive into the colaboratory

[6]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

0.4 Importing dataset directories

[7]:
```
from os import getcwd , listdir
from os.path import join
curr_dir = getcwd()
drive_dir = join(curr_dir , 'drive','MyDrive')
dataset_dir = join(drive_dir,'ChallengingExpt')
```

[8]:
```
listdir(dataset_dir)
```

[8]:
```
['Malayalam_dev_1-5_cleaned.csv',
 'Malayalam_test_1-5_cleaned.csv',
 'Malayalam_train_1-5_cleaned.csv',
 '.ipynb_checkpoints']
```

0.5 Loading training data

[9]:
```
train_df_path = join(dataset_dir,'Malayalam_train_1-5_cleaned.csv')
# Load the dataset into a pandas dataframe.
#train_df = pd.read_csv(train_df_path)
train_df = pd.read_csv(train_df_path, delimiter='\t', header=None,␣
 ↪names=['text', 'label'])
# Report the number of sentences.
print('Number of training sentences: {:,}\n'.format(train_df.shape[0]))

# Display 10 random rows from the data.
train_df.sample(10)
```

```
Number of training sentences: 10,804
```

[9]:
```
                                                     text  \
9855              Lal fans dis like adi ippozhe thudangi
7254                 …          …
6847
9995                                 …
5833             Nammude Dileep chettane  kodukilla like
10478  Madhuraraja pottum ennu orappula ettan fans iv…
6232                           dialogu…
9733   Ippam mazha kalamayond mannira nallanam kuthik…
4100   Mavane kola ghandla irukan kollama vidamaatan …
33     Shariya killi chundan thanne sreenivasan enthe…

                                 label
9855   Offensive_Targeted_Insult_Group
7254                     Not_offensive
6847                     Not_offensive
9995                     Not_offensive
5833                     Not_offensive
10478                    Not_offensive
6232                     Not_offensive
9733                     Not_offensive
4100                     Not_offensive
33                       Not_offensive
```

[10]:
```python
train_df['label'].value_counts()
```

[10]:
```
label
Not_offensive                             10374
Offensive_Targeted_Insult_Individual        170
Offensive_Untargetede                       154
Offensive_Targeted_Insult_Group             105
category                                      1
Name: count, dtype: int64
```

[11]:
```python
le = LabelEncoder()
train_df['label'] = le.fit_transform(train_df['label'])
```

[12]:
```python
train_df['label'].value_counts()
```

[12]:
```
label
0    10374
2      170
3      154
1      105
4        1
Name: count, dtype: int64
```

0.5.1 Loading train_sentences and train_labels

[13]:
```
# Get the lists of sentences and their labels.
train_sentences = train_df['text'].values
train_labels = train_df['label'].values
```

0.6 Saving Class Names

[14]:
```
class_names = [
                ␣
  ↪'Not_offensive','Offensive_Targeted_Insult_Group','Offensive_Targeted_Insult_Individual',
                'Offensive_Untargetede','not-malayalam'
]
```

0.7 Loading the validation data

[15]:
```
val_df_path = join(dataset_dir,'Malayalam_dev_1-5_cleaned.csv')
# Load the dataset into a pandas dataframe.
# val_df = pd.read_csv(val_df_path)
val_df = pd.read_csv(val_df_path, delimiter='\t', header=None, names=['text',␣
  ↪'label'])
# Report the number of sentences.
print('Number of training sentences: {:,}\n'.format(val_df.shape[0]))

# Display 10 random rows from the data.
val_df.sample(10)
```

```
Number of training sentences: 1,766
```

[15]:
```
                                                    text  \
376   Look aayi lalettan thonni prithivi muthaanu li…
498        Trendingil 27   Evide ikka fans Come on fans
1228    Eni aa pattukalokke upload cheythu kolamakkalle
511   Odiyante climaxil mohanlal marikkumennu ariyaa…
584                                             KOMALI
1024  Chakochan carrier break avum enn       a…
35    Ettante records pottikkan ettan mathrame ollu …
1254  uff trailer  mamuka mohanlal legends aanu oru …
590                            .But   …
1730  Anthamaaya thaara aaradana aaswadana nilavara …

                                  label
376                       Not_offensive
498                       Not_offensive
1228                      Not_offensive
511                       Not_offensive
```

```
584     Offensive_Targeted_Insult_Individual
1024                        Not_offensive
35                          Not_offensive
1254                        Not_offensive
590                         Not_offensive
1730                        Not_offensive
```

[16]:
```
val_df['label'].value_counts()
```

[16]:
```
label
Not_offensive                           1709
Offensive_Targeted_Insult_Individual      23
Offensive_Untargetede                     20
Offensive_Targeted_Insult_Group           13
category                                   1
Name: count, dtype: int64
```

[17]:
```
val_df['label'] = le.transform(val_df['label'])
```

[18]:
```
val_df['label'].value_counts()
```

[18]:
```
label
0    1709
2      23
3      20
1      13
4       1
Name: count, dtype: int64
```

0.7.1 Loading val_sentences and val_labels

[19]:
```
# Get the lists of sentences and their labels.
val_sentences = val_df['text'].values
val_labels = val_df['label'].values
```

0.8 Handling class imbalance using sklearn's compute_class_weight

[20]:
```
#compute the class weights
class_wts = compute_class_weight('balanced', classes=np.unique(train_labels),␣
 ↪y=train_labels)

print(class_wts)
```

```
[2.08289956e-01 2.05790476e+01 1.27105882e+01 1.40311688e+01
 2.16080000e+03]
```

0.9 Helper Functions

1) Update Interval Function

[21]:
```python
def good_update_interval(total_iters, num_desired_updates):
    '''
    This function will try to pick an intelligent progress update interval
    based on the magnitude of the total iterations.

    Parameters:
      `total_iters` - The number of iterations in the for-loop.
      `num_desired_updates` - How many times we want to see an update over the
                              course of the for-loop.
    '''
    # Divide the total iterations by the desired number of updates. Most likely
    # this will be some ugly number.
    exact_interval = total_iters / num_desired_updates

    # The `round` function has the ability to round down a number to, e.g., the
    # nearest thousandth: round(exact_interval, -3)
    #
    # To determine the magnitude to round to, find the magnitude of the total,
    # and then go one magnitude below that.

    # Get the order of magnitude of the total.
    order_of_mag = len(str(total_iters)) - 1

    # Our update interval should be rounded to an order of magnitude smaller.
    round_mag = order_of_mag - 1

    # Round down and cast to an int.
    update_interval = int(round(exact_interval, -round_mag))

    # Don't allow the interval to be zero!
    if update_interval == 0:
        update_interval = 1

    return update_interval
```

2) Format time function

[22]:
```python
def format_time(elapsed):
    '''
    Takes a time in seconds and returns a string hh:mm:ss
    '''
    # Round to the nearest second.
    elapsed_rounded = int(round((elapsed)))

    # Format as hh:mm:ss
```

```
    return str(datetime.timedelta(seconds=elapsed_rounded))
```

3) Make Smart Batches function

[23]:
```
def make_smart_batches(text_samples, labels,␣
↪batch_size,tokenizer,max_input_length):
    '''
    This function combines all of the required steps to prepare batches.
    '''

    print('Creating Smart Batches from {:,} examples with batch size {:,}...\n'.
↪format(len(text_samples), batch_size))

    # =========================
    #   Tokenize & Truncate
    # =========================

    full_input_ids = []

    # Tokenize all training examples
    print('Tokenizing {:,} samples...'.format(len(labels)))

    # Choose an interval on which to print progress updates.
    update_interval = good_update_interval(total_iters=len(labels),␣
↪num_desired_updates=10)

    # For each training example...
    for text in text_samples:

        # Report progress.
        if ((len(full_input_ids) % update_interval) == 0):
            print('  Tokenized {:,} samples.'.format(len(full_input_ids)))

        # Tokenize the sample.
        input_ids = tokenizer.encode(text=text,              # Text to encode.
                                    add_special_tokens=True, # Do add specials.
                                    max_length=max_input_length,
                                    truncation=True,         # Do Truncate!
                                    padding=False)           # DO NOT pad.

        # Add the tokenized result to our list.
        full_input_ids.append(input_ids)

    print('DONE.')
    print('{:>10,} samples\n'.format(len(full_input_ids)))

    # =========================
```

```
    #      Select Batches
    # ==========================

    # Sort the two lists together by the length of the input sequence.
    samples = sorted(zip(full_input_ids, labels), key=lambda x: len(x[0]))

    print('{:>10,} samples after sorting\n'.format(len(samples)))

    import random

    # List of batches that we'll construct.
    batch_ordered_sentences = []
    batch_ordered_labels = []

    print('Creating batches of size {:}...'.format(batch_size))

    # Choose an interval on which to print progress updates.
    update_interval = good_update_interval(total_iters=len(samples),␣
↪num_desired_updates=10)

    # Loop over all of the input samples...
    while len(samples) > 0:

        # Report progress.
        if ((len(batch_ordered_sentences) % update_interval) == 0 \
            and not len(batch_ordered_sentences) == 0):
            print('  Selected {:,} batches.'.
↪format(len(batch_ordered_sentences)))

        # `to_take` is our actual batch size. It will be `batch_size` until
        # we get to the last batch, which may be smaller.
        to_take = min(batch_size, len(samples))

        # Pick a random index in the list of remaining samples to start
        # our batch at.
        select = random.randint(0, len(samples) - to_take)

        # Select a contiguous batch of samples starting at `select`.
        #print("Selecting batch from {:} to {:}".format(select, select+to_take))
        batch = samples[select:(select + to_take)]

        #print("Batch length:", len(batch))

        # Each sample is a tuple--split them apart to create a separate list of
        # sequences and a list of labels for this batch.
        batch_ordered_sentences.append([s[0] for s in batch])
        batch_ordered_labels.append([s[1] for s in batch])
```

```
        # Remove these samples from the list.
        del samples[select:select + to_take]

    print('\n  DONE - Selected {:,} batches.\n'.
↪format(len(batch_ordered_sentences)))

    # =========================
    #        Add Padding
    # =========================

    print('Padding out sequences within each batch...')

    py_inputs = []
    py_attn_masks = []
    py_labels = []

    # For each batch...
    for (batch_inputs, batch_labels) in zip(batch_ordered_sentences,␣
↪batch_ordered_labels):

        # New version of the batch, this time with padded sequences and now with
        # attention masks defined.
        batch_padded_inputs = []
        batch_attn_masks = []

        # First, find the longest sample in the batch.
        # Note that the sequences do currently include the special tokens!
        max_size = max([len(sen) for sen in batch_inputs])

        # For each input in this batch...
        for sen in batch_inputs:

            # How many pad tokens do we need to add?
            num_pads = max_size - len(sen)

            # Add `num_pads` padding tokens to the end of the sequence.
            padded_input = sen + [tokenizer.pad_token_id]*num_pads

            # Define the attention mask--it's just a `1` for every real token
            # and a `0` for every padding token.
            attn_mask = [1] * len(sen) + [0] * num_pads

            # Add the padded results to the batch.
            batch_padded_inputs.append(padded_input)
            batch_attn_masks.append(attn_mask)
```

```
            # Our batch has been padded, so we need to save this updated batch.
            # We also need the inputs to be PyTorch tensors, so we'll do that here.
            # Todo - Michael's code specified "dtype=torch.long"
            py_inputs.append(torch.tensor(batch_padded_inputs))
            py_attn_masks.append(torch.tensor(batch_attn_masks))
            py_labels.append(torch.tensor(batch_labels))

    print('  DONE.')

    # Return the smart-batched dataset!
    return (py_inputs, py_attn_masks, py_labels)
```

0.9.1 4) Make Smart Batches On Test Set (Without labels)

[24]:
```
def make_smart_batches_on_test(text_samples, text_ids,␣
↪batch_size,tokenizer,max_input_length):
    '''
    This function combines all of the required steps to prepare batches.
    '''

    print('Creating Smart Batches from {:,} examples with batch size {:,}...\n'.
↪format(len(text_samples), batch_size))

    # =========================
    #   Tokenize & Truncate
    # =========================

    full_input_ids = []

    # Tokenize all training examples
    print('Tokenizing {:,} samples...'.format(len(text_samples)))

    # Choose an interval on which to print progress updates.
    update_interval = good_update_interval(total_iters=len(text_samples),␣
↪num_desired_updates=10)

    # For each training example...
    for text in text_samples:

        # Report progress.
        if ((len(full_input_ids) % update_interval) == 0):
            print('  Tokenized {:,} samples.'.format(len(full_input_ids)))

        # Tokenize the sample.
        input_ids = tokenizer.encode(text=text,              # Text to encode.
                                    add_special_tokens=True, # Do add specials.
```

```
                                        max_length=max_input_length,
                                        truncation=True,        # Do Truncate!
                                        padding=False)          # DO NOT pad.

        # Add the tokenized result to our list.
        full_input_ids.append(input_ids)

    print('DONE.')
    print('{:>10,} samples\n'.format(len(full_input_ids)))

    # =========================
    #      Select Batches
    # =========================

    # Sort the two lists together by the length of the input sequence.
    samples = sorted(zip(full_input_ids, text_ids), key=lambda x: len(x[0]))

    print('{:>10,} samples after sorting\n'.format(len(samples)))

    import random

    # List of batches that we'll construct.
    batch_ordered_sentences = []
    batch_ordered_ids = []

    print('Creating batches of size {:}...'.format(batch_size))

    # Choose an interval on which to print progress updates.
    update_interval = good_update_interval(total_iters=len(samples),␣
↪num_desired_updates=10)

    # Loop over all of the input samples...
    while len(samples) > 0:

        # Report progress.
        if ((len(batch_ordered_sentences) % update_interval) == 0 \
            and not len(batch_ordered_sentences) == 0):
            print('  Selected {:,} batches.'.
↪format(len(batch_ordered_sentences)))

        # `to_take` is our actual batch size. It will be `batch_size` until
        # we get to the last batch, which may be smaller.
        to_take = min(batch_size, len(samples))

        # Pick a random index in the list of remaining samples to start
        # our batch at.
        select = random.randint(0, len(samples) - to_take)
```

```
        # Select a contiguous batch of samples starting at `select`.
        #print("Selecting batch from {:} to {:}".format(select, select+to_take))
        batch = samples[select:(select + to_take)]

        #print("Batch length:", len(batch))

        # Each sample is a tuple--split them apart to create a separate list of
        # sequences and a list of labels for this batch.
        batch_ordered_sentences.append([s[0] for s in batch])
        batch_ordered_ids.append([s[1] for s in batch])

        # Remove these samples from the list.
        del samples[select:select + to_take]

    print('\n  DONE - Selected {:,} batches.\n'.
 ↪format(len(batch_ordered_sentences)))

    # =========================
    #       Add Padding
    # =========================

    print('Padding out sequences within each batch...')

    py_inputs = []
    py_attn_masks = []
    py_ids = []

    # For each batch...
    for (batch_inputs, batch_ids) in zip(batch_ordered_sentences,␣
 ↪batch_ordered_ids):

        # New version of the batch, this time with padded sequences and now with
        # attention masks defined.
        batch_padded_inputs = []
        batch_attn_masks = []

        # First, find the longest sample in the batch.
        # Note that the sequences do currently include the special tokens!
        max_size = max([len(sen) for sen in batch_inputs])

        # For each input in this batch...
        for sen in batch_inputs:

            # How many pad tokens do we need to add?
            num_pads = max_size - len(sen)
```

```
            # Add `num_pads` padding tokens to the end of the sequence.
            padded_input = sen + [tokenizer.pad_token_id]*num_pads

            # Define the attention mask--it's just a `1` for every real token
            # and a `0` for every padding token.
            attn_mask = [1] * len(sen) + [0] * num_pads

            # Add the padded results to the batch.
            batch_padded_inputs.append(padded_input)
            batch_attn_masks.append(attn_mask)

        # Our batch has been padded, so we need to save this updated batch.
        # We also need the inputs to be PyTorch tensors, so we'll do that here.
        # Todo - Michael's code specified "dtype=torch.long"
        py_inputs.append(torch.tensor(batch_padded_inputs))
        py_attn_masks.append(torch.tensor(batch_attn_masks))
        py_ids.append(torch.tensor(batch_ids))

    print('  DONE.')

    # Return the smart-batched dataset!
    return (py_inputs, py_attn_masks, py_ids)
```

0.9.2 5) Function for plotting training history

[25]:
```
def plot_training_history(history):
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))

  ax1.plot(history['train_loss'], label='train loss')
  ax1.plot(history['val_loss'], label='validation loss')

  ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
  ax1.set_ylim([-0.05, 1.05])
  ax1.legend()
  ax1.set_ylabel('Loss')
  ax1.set_xlabel('Epoch')

  ax2.plot(history['train_acc'], label='train accuracy')
  ax2.plot(history['val_acc'], label='validation accuracy')

  ax2.xaxis.set_major_locator(MaxNLocator(integer=True))
  ax2.set_ylim([-0.05, 1.05])
  ax2.legend()

  ax2.set_ylabel('Accuracy')
  ax2.set_xlabel('Epoch')
```

```
fig.suptitle('Training history')
```

0.9.3 6) Function to check accuracy

```
[26]: def check_accuracy(predictions,true_labels):
        # Combine the results across the batches.
        predictions = np.concatenate(predictions, axis=0)
        true_labels = np.concatenate(true_labels, axis=0)

        # Choose the label with the highest score as our prediction.
        preds = np.argmax(predictions, axis=1).flatten()

        # Calculate simple flat accuracy -- number correct over total number.
        accuracy = (preds == true_labels).mean()

        return accuracy
```

0.9.4 7) Function to evaluate model

```
[27]: def eval_model(model,py_inputs, py_attn_masks, py_labels):
      # Prediction on test set
        t0 = time.time()

        print('Doing validation on {:,} sentences...'.format(len(py_labels)))

        # Put model in evaluation mode
        model.eval()

        # Tracking variables
        predictions , true_labels = [], []

        # Choose an interval on which to print progress updates.
        update_interval = good_update_interval(total_iters=len(py_inputs),␣
       ↪num_desired_updates=10)

        # Measure elapsed time.
        t0 = time.time()
        total_val_loss = 0

        # Put model in evaluation mode
        model.eval()

        # For each batch of training data...
        for step in range(0, len(py_inputs)):

            # Progress update every 100 batches.
```

```
        if step % update_interval == 0 and not step == 0:
            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)

            # Calculate the time remaining based on our progress.
            steps_per_sec = (time.time() - t0) / step
            remaining_sec = steps_per_sec * (len(py_inputs) - step)
            remaining = format_time(remaining_sec)

            # Report progress.
            print('  Batch {:>7,}  of  {:>7,}.    Elapsed: {:}.  Remaining: {:}'.
↪format(step, len(py_inputs), elapsed, remaining))

        # Copy the batch to the GPU.
        b_input_ids = py_inputs[step].to(device)
        b_input_mask = py_attn_masks[step].to(device)
        b_labels = py_labels[step].to(device)

        # Telling the model not to compute or store gradients, saving memory and
        # speeding up prediction
        with torch.no_grad():
          output = model(b_input_ids,
                                 token_type_ids=None,
                                 attention_mask=b_input_mask,
                                 labels=b_labels)


        logits = output.logits
        loss = output.loss
        # Move logits and labels to CPU
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()

        # Store predictions and true labels
        predictions.append(logits)
        true_labels.append(label_ids)

        total_val_loss += loss.item()

    val_accuracy = check_accuracy(predictions,true_labels)

    validation_time = format_time(time.time() - t0)

    # Calculate the average loss over all of the batches.
    avg_val_loss = total_val_loss / len(py_inputs)
```

```
    # print('Accuracy: {:.3f}'.format(val_accuracy))
    return (avg_val_loss,val_accuracy,validation_time)
```

0.9.5 8) Function for making predictions on our test dataset

[28]:
```
def get_predictions(py_inputs, py_attn_masks, py_labels):

    print('Predicting labels for {:,} test batches...'.format(len(py_labels)))

    # Put model in evaluation mode
    model.eval()

    # Tracking variables
    predictions , true_labels = [], []

    # Choose an interval on which to print progress updates.
    update_interval = good_update_interval(total_iters=len(py_inputs),␣
↪num_desired_updates=10)

    # Measure elapsed time.
    t0 = time.time()

    # Put model in evaluation mode
    model.eval()

    # For each batch of training data...
    for step in range(0, len(py_inputs)):

        # Progress update every 100 batches.
        if step % update_interval == 0 and not step == 0:
            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)

            # Calculate the time remaining based on our progress.
            steps_per_sec = (time.time() - t0) / step
            remaining_sec = steps_per_sec * (len(py_inputs) - step)
            remaining = format_time(remaining_sec)

            # Report progress.
            print('  Batch {:>7,}  of  {:>7,}.    Elapsed: {:}.  Remaining: {:}'.
↪format(step, len(py_inputs), elapsed, remaining))

        # Copy the batch to the GPU.
        b_input_ids = py_inputs[step].to(device)
        b_input_mask = py_attn_masks[step].to(device)
        b_labels = py_labels[step].to(device)
```

```
    # Telling the model not to compute or store gradients, saving memory and
    # speeding up prediction
    with torch.no_grad():
        # Forward pass, calculate logit predictions
        outputs = model(b_input_ids, b_input_mask)

    logits = outputs.logits

    # Move logits and labels to CPU
    logits = logits.detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()

    # Store predictions and true labels
    predictions.append(logits)
    true_labels.append(label_ids)
    # Combine the results across the batches.
predictions = np.concatenate(predictions, axis=0)
true_labels = np.concatenate(true_labels, axis=0)

    # Choose the label with the highest score as our prediction.
preds = np.argmax(predictions, axis=1).flatten()

return (preds,true_labels)
```

0.9.6 9) Function for making predictions on test dataset(without labels)

[29]:
```
def get_predictions_test(py_inputs, py_attn_masks,py_ids):

  print('Predicting labels for {:,} test batches...'.format(len(py_inputs)))

  # Put model in evaluation mode
  model.eval()

  # Tracking variables
  predictions , true_labels ,ids = [], [] , []

  # Choose an interval on which to print progress updates.
  update_interval = good_update_interval(total_iters=len(py_inputs),␣
↪num_desired_updates=10)

  # Measure elapsed time.
  t0 = time.time()

  # Put model in evaluation mode
  model.eval()
```

```python
# For each batch of training data...
for step in range(0, len(py_inputs)):

    # Progress update every 100 batches.
    if step % update_interval == 0 and not step == 0:
        # Calculate elapsed time in minutes.
        elapsed = format_time(time.time() - t0)

        # Calculate the time remaining based on our progress.
        steps_per_sec = (time.time() - t0) / step
        remaining_sec = steps_per_sec * (len(py_inputs) - step)
        remaining = format_time(remaining_sec)

        # Report progress.
        print('  Batch {:>7,}  of  {:>7,}.    Elapsed: {:}.  Remaining: {:}'.
↪format(step, len(py_inputs), elapsed, remaining))

    # Copy the batch to the GPU.
    b_input_ids = py_inputs[step].to(device)
    b_input_mask = py_attn_masks[step].to(device)
    b_ids = py_ids[step].to(device)

    # Telling the model not to compute or store gradients, saving memory and
    # speeding up prediction
    with torch.no_grad():
        # Forward pass, calculate logit predictions
        outputs = model(b_input_ids, b_input_mask)

    logits = outputs.logits

    # Move logits and labels to CPU
    logits = logits.detach().cpu().numpy()
    b_ids = b_ids.detach().cpu().numpy()
    # Store predictions
    predictions.append(logits)
    ids.append(b_ids)
    # Combine the results across the batches.
predictions = np.concatenate(predictions, axis=0)
ids = np.concatenate(ids,axis=0)
# Choose the label with the highest score as our prediction.
preds = np.argmax(predictions, axis=1).flatten()

return ids,preds
```

0.9.7 10) Confusion Matrix Helper Function

[30]:
```
def show_confusion_matrix(confusion_matrix, class_names):

  cm = confusion_matrix.copy()

  cell_counts = cm.flatten()

  cm_row_norm = cm / cm.sum(axis=1)[:, np.newaxis]

  row_percentages = ["{0:.2f}".format(value) for value in cm_row_norm.flatten()]

  cell_labels = [f"{cnt}\n{per}" for cnt, per in zip(cell_counts,␣
↪row_percentages)]
  cell_labels = np.asarray(cell_labels).reshape(cm.shape[0], cm.shape[1])

  df_cm = pd.DataFrame(cm_row_norm, index=class_names, columns=class_names)

  hmap = sns.heatmap(df_cm, annot=cell_labels, fmt="", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30,␣
↪ha='right')
  plt.ylabel('True Sign')
  plt.xlabel('Predicted Sign');
```

0.10 Smart Batching of the training data

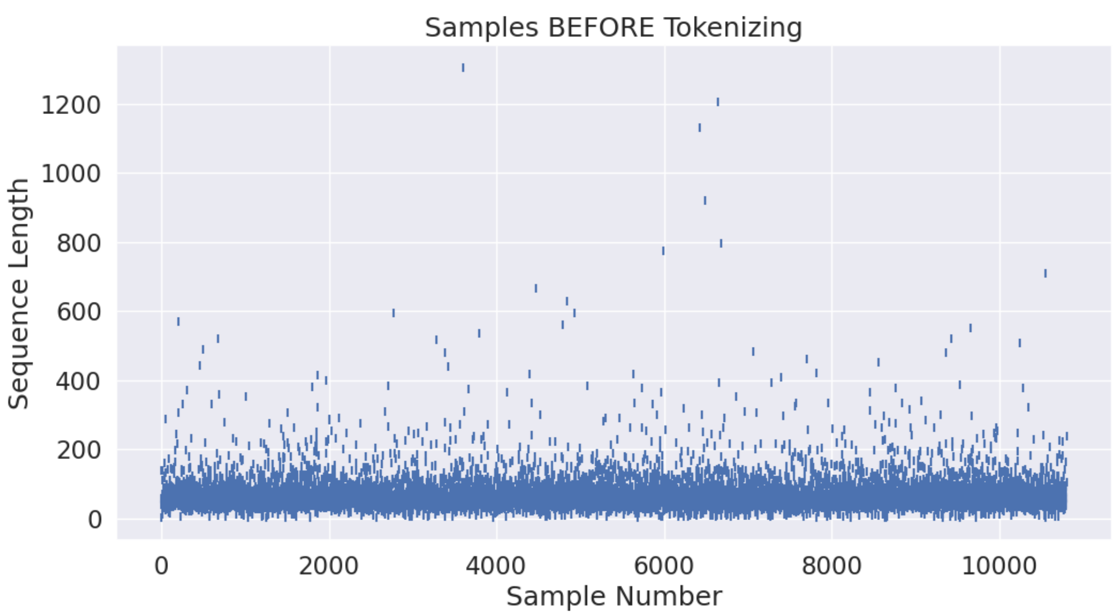
[31]:
```
model_name = 'xlm-roberta-base'
# Load the BERT tokenizer.
print(f'Loading {model_name} tokenizer...')
tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=False)
```

```
Loading xlm-roberta-base tokenizer…

/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(

tokenizer_config.json:   0%|          | 0.00/25.0 [00:00<?, ?B/s]

config.json:   0%|          | 0.00/615 [00:00<?, ?B/s]
```

```
sentencepiece.bpe.model:   0%|          | 0.00/5.07M [00:00<?, ?B/s]

tokenizer.json:   0%|          | 0.00/9.10M [00:00<?, ?B/s]
```

[32]:
```
lengths = []
for text in train_sentences:
  lengths.append(len(text))
```

[33]:
```
# Use plot styling from seaborn.
sns.set(style='darkgrid')

# Increase the plot size and font size.
sns.set(font_scale=1.5)
plt.rcParams["figure.figsize"] = (12,6)

plt.scatter(range(0, len(lengths)), lengths, marker="|")

plt.xlabel('Sample Number')
plt.ylabel('Sequence Length')
plt.title('Samples BEFORE Tokenizing')

plt.show()
```



0.10.1 Tokenizing the sequences

[34]:
```
max_input_length = 400
```

[35]:

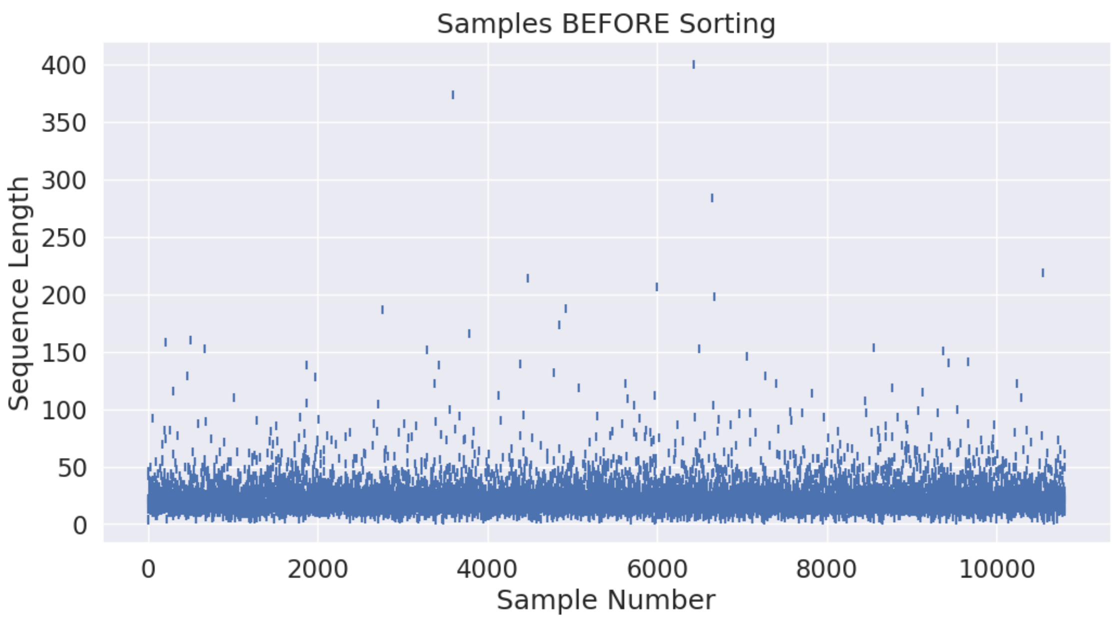
```
full_input_ids = []
labels = []

# Tokenize all training examples
print('Tokenizing {:,} training samples...'.format(len(train_sentences)))

# Choose an interval on which to print progress updates.
update_interval = good_update_interval(total_iters=len(train_sentences),␣
↪num_desired_updates=10)

# For each training example...
for text in train_sentences:

    # Report progress.
    if ((len(full_input_ids) % update_interval) == 0):
        print('  Tokenized {:,} samples.'.format(len(full_input_ids)))

    # Tokenize the sentence.
    input_ids = tokenizer.encode(text=text,
                                 add_special_tokens=True,
                                 max_length=max_input_length,
                                 truncation=True,
                                 padding=False)

    # Add the tokenized result to our list.
    full_input_ids.append(input_ids)

print('DONE.')
print('{:>10,} samples'.format(len(full_input_ids)))
```

```
Tokenizing 10,804 training samples…
  Tokenized 0 samples.
  Tokenized 1,000 samples.
  Tokenized 2,000 samples.
  Tokenized 3,000 samples.
  Tokenized 4,000 samples.
  Tokenized 5,000 samples.
  Tokenized 6,000 samples.
  Tokenized 7,000 samples.
  Tokenized 8,000 samples.
  Tokenized 9,000 samples.
  Tokenized 10,000 samples.
DONE.
    10,804 samples
```

[36]:

```
# Get all of the lengths.
unsorted_lengths = [len(x) for x in full_input_ids]
```

[37]:
```python
# Use plot styling from seaborn.
sns.set(style='darkgrid')

# Increase the plot size and font size.
sns.set(font_scale=1.5)
plt.rcParams["figure.figsize"] = (12,6)

plt.scatter(range(0, len(unsorted_lengths)), unsorted_lengths, marker="|")

plt.xlabel('Sample Number')
plt.ylabel('Sequence Length')
plt.title('Samples BEFORE Sorting')

plt.show()
```



[38]:
```python
# Sort the two lists together by the length of the input sequence.
train_samples = sorted(zip(full_input_ids, train_labels), key=lambda x:␣
↪len(x[0]))
```

[39]:
```python
train_samples[0:5]
```

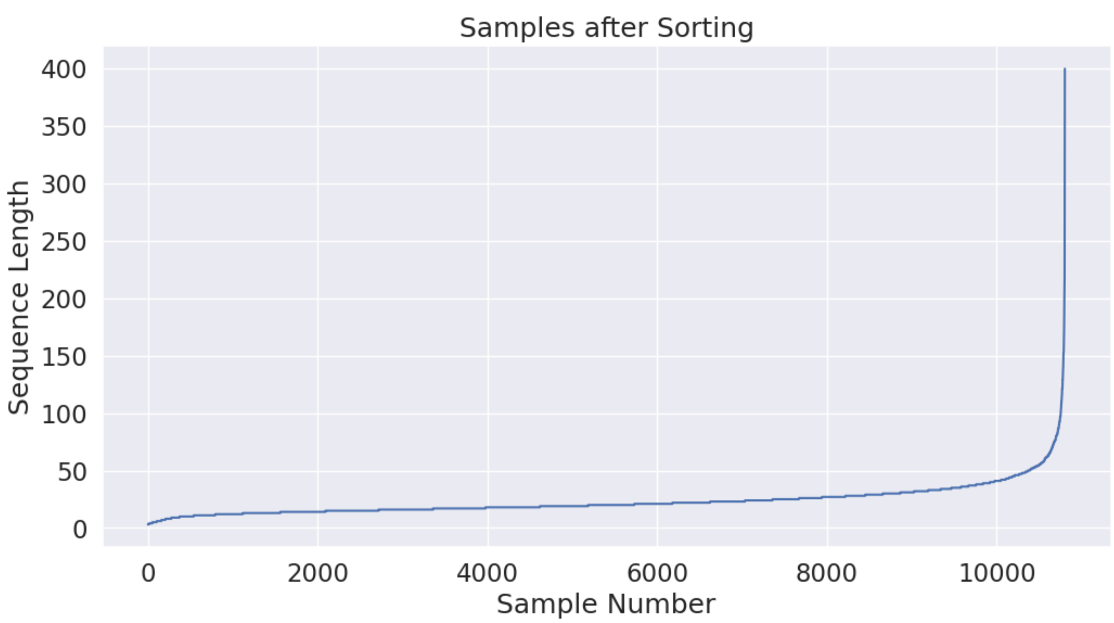
[39]:
```
[([0, 7986, 2], 4),
 ([0, 12911, 2], 0),
 ([0, 145200, 2], 0),
 ([0, 8414, 2], 0),
 ([0, 158618, 2], 0)]
```

[40]:
```
print('Shortest sample:', len(train_samples[0][0]))
print('Longest sample:', len(train_samples[-1][0]))
```

```
Shortest sample: 3
Longest sample: 400
```

[41]:
```
# Get the new list of lengths after sorting.
sorted_lengths = [len(s[0]) for s in train_samples]
```

[42]:
```
plt.plot(range(0, len(sorted_lengths)), sorted_lengths)

plt.xlabel('Sample Number')
plt.ylabel('Sequence Length')
plt.title('Samples after Sorting')

plt.show()
```



0.10.2 Random Batch Selections

[43]:
```
train_samples[0]
```

[43]:
```
([0, 7986, 2], 4)
```

[44]:
```
batch_size = 16
```

[45]:
```
# List of batches that we'll construct.
batch_ordered_sentences = []
batch_ordered_labels = []

print('Creating training batches of size {:}'.format(batch_size))

# Loop over all of the input samples...
while len(train_samples) > 0:

    # Report progress.
    if ((len(batch_ordered_sentences) % 100) == 0):
        print('  Selected {:,} batches.'.format(len(batch_ordered_sentences)))

    # `to_take` is our actual batch size. It will be `batch_size` until
    # we get to the last batch, which may be smaller.
    to_take = min(batch_size, len(train_samples))

    # Pick a random index in the list of remaining samples to start
    # our batch at.
    select = random.randint(0, len(train_samples) - to_take)

    # Select a contiguous batch of samples starting at `select`.
    batch = train_samples[select:(select + to_take)]

    # Each sample is a tuple--split them apart to create a separate list of
    # sequences and a list of labels for this batch.
    batch_ordered_sentences.append([s[0] for s in batch])
    batch_ordered_labels.append([s[1] for s in batch])

    # Remove these samples from the list.
    del train_samples[select:select + to_take]

print('\n  DONE - {:,} batches.'.format(len(batch_ordered_sentences)))
```

```
Creating training batches of size 16
  Selected 0 batches.
  Selected 100 batches.
  Selected 200 batches.
  Selected 300 batches.
  Selected 400 batches.
  Selected 500 batches.
  Selected 600 batches.

  DONE - 676 batches.
```

[46]:
```
print(batch_ordered_sentences[0])
```

```
[[0, 57719, 15626, 3463, 9281, 99916, 10718, 171363, 18905, 26641, 5, 5, 12263,
```

115672, 204557, 187175, 23277, 1763, 12650, 44122, 2599, 2282, 172468, 574, 28, 33730, 538, 89289, 100, 46951, 4493, 15771, 3637, 46951, 4493, 4492, 10222, 34989, 211405, 847, 118983, 9979, 3579, 209037, 10051, 22522, 12331, 6, 44174, 144904, 20411, 574, 27, 2], [0, 24799, 24005, 19653, 6, 65120, 121, 19, 28866, 2755, 10713, 842, 46892, 60055, 70, 9022, 28866, 8201, 19, 1468, 943, 8043, 289, 5568, 432, 685, 161, 35992, 2412, 416, 72033, 121, 20470, 19, 539, 14594, 594, 51488, 14849, 15467, 153, 5, 8015, 647, 23373, 104105, 839, 16301, 6, 5, 128505, 284, 129138, 2], [0, 357, 18369, 34, 173, 1704, 152713, 11, 456, 1861, 1098, 32716, 2154, 309, 539, 82, 4505, 34, 2522, 7328, 173, 331, 156, 8848, 119255, 31, 2589, 1507, 694, 34, 5799, 331, 74552, 59206, 21441, 8962, 2198, 5799, 11439, 5164, 22462, 21441, 14, 55101, 178598, 11461, 60674, 47707, 10, 28929, 6332, 669, 195, 2], [0, 136984, 344, 2412, 7, 3915, 484, 10902, 416, 20981, 160, 2387, 5786, 1436, 1961, 92, 36045, 1645, 4450, 1436, 59724, 2412, 33806, 116, 8517, 160, 8805, 128, 5548, 173, 7207, 771, 685, 316, 104035, 143, 24, 169, 26372, 1153, 9233, 56203, 13315, 5, 79573, 18926, 21, 133, 16883, 756, 70, 2965, 5, 2], [0, 5539, 17093, 5420, 3338, 50359, 120777, 30141, 15752, 214354, 5, 6177, 17093, 90937, 6753, 44953, 196264, 36862, 2807, 4668, 1614, 34601, 4295, 72465, 9664, 59382, 181472, 111509, 3435, 8210, 22522, 6, 5, 1763, 88816, 4934, 3072, 70920, 220218, 6, 226109, 18896, 10229, 51435, 9554, 5, 22362, 5539, 17093, 5420, 3338, 50359, 120777, 2], [0, 88985, 85152, 3250, 59899, 3338, 124883, 58018, 112974, 40189, 14133, 36544, 3463, 153, 77048, 218519, 11079, 140961, 456, 75161, 5, 5, 228422, 847, 1468, 56161, 22030, 91889, 6177, 92989, 77976, 62768, 153, 48705, 9973, 2117, 6014, 113024, 153, 141981, 174257, 6, 113024, 70920, 22562, 9067, 18356, 27795, 6928, 53606, 153, 618, 425, 2], [0, 142107, 21944, 28331, 6177, 85298, 76557, 2117, 208484, 6177, 23768, 11300, 13970, 15553, 4376, 18208, 38234, 25835, 25239, 16201, 18002, 1763, 190265, 4492, 86027, 6177, 23768, 12194, 13970, 23587, 3254, 2117, 232355, 7358, 59382, 6, 88337, 3338, 28318, 181386, 191139, 102704, 847, 214287, 49070, 2845, 8265, 50210, 16056, 4376, 18208, 13427, 12267, 2], [0, 241, 27, 353, 1648, 34310, 2310, 20981, 10, 147, 9233, 26286, 331, 1679, 22404, 66, 32993, 56406, 1165, 653, 2864, 27, 357, 6236, 7895, 2387, 50573, 80705, 53, 685, 11263, 67594, 653, 2864, 20655, 4776, 771, 36121, 27, 1519, 693, 200, 693, 200, 693, 46889, 1992, 206, 879, 10632, 429, 5, 27, 2], [0, 109303, 141383, 26641, 23102, 144822, 12681, 119377, 20232, 3072, 23694, 166315, 20232, 233466, 23833, 8126, 82425, 13795, 110498, 5, 27, 149270, 3637, 18208, 4668, 619, 122251, 5, 26641, 151982, 574, 37702, 1230, 7612, 434, 26641, 23833, 2381, 43702, 99150, 69220, 8463, 17334, 8126, 5903, 8527, 23694, 166315, 20232, 228008, 43515, 5, 5, 2], [0, 106023, 18785, 6925, 10, 41066, 513, 31, 1743, 2591, 33721, 139603, 66, 10015, 34, 21642, 18, 44462, 13, 5, 27, 13474, 141, 27, 25740, 594, 23490, 136, 172376, 1185, 319, 21642, 8805, 112, 5, 27, 4776, 18023, 7665, 4776, 3501, 86, 86, 5, 27, 14, 8805, 6, 4160, 119871, 27, 3337, 1033, 2], [0, 3023, 24135, 1666, 43735, 31113, 5539, 17093, 5420, 3338, 114797, 80296, 30141, 15752, 3338, 146817, 5, 5, 18915, 146308, 3254, 7807, 3637, 2569, 31794, 10718, 231463, 11738, 26641, 86027, 5, 5, 20038, 8463, 71236, 42657, 2599, 46951, 7327, 6368, 102704, 847, 18896, 2141, 13884, 46951, 7327, 2569, 15843, 2210, 49559, 5, 5, 2], [0, 72391, 7807, 3637, 6, 95203, 6, 65746, 2381, 24720, 35874, 20489, 73, 944, 30476, 115004, 5942, 3435, 56670, 139817, 187920, 6177, 36544, 123325, 5674, 8208, 27693, 3254, 65073, 3023, 121678, 163, 206, 123325, 3929, 72033, 468, 12724,

```
16300, 282, 206, 468, 724, 6328, 238, 284, 24401, 468, 7, 418, 594, 181596, 505,
2], [0, 6, 185279, 10331, 6, 185279, 10331, 47812, 115962, 47812, 115962, 6014,
13427, 105674, 2381, 32, 18166, 7807, 10222, 8208, 27693, 3254, 3463, 2569,
1884, 13798, 7807, 24720, 18779, 3612, 60329, 3463, 2569, 15843, 45839, 15098,
3463, 129978, 38145, 28392, 5, 468, 111836, 4044, 14, 7228, 23, 6, 42072, 468,
143, 1974, 1076, 2], [0, 468, 2141, 2903, 13234, 159211, 69635, 115962, 7059,
12274, 102054, 5, 5, 125851, 2381, 26805, 101103, 11300, 33404, 5, 5, 1763,
24617, 97289, 132584, 3463, 2210, 195042, 2381, 138287, 5, 5, 468, 7, 3038,
21135, 11075, 1639, 15513, 10854, 62, 79347, 62, 110500, 23373, 11766, 6,
116387, 62, 139645, 3637, 619, 56812, 2], [0, 53884, 23247, 101932, 2559,
176268, 33113, 164911, 47812, 6263, 7358, 68243, 3023, 5903, 164911, 202456,
7358, 30476, 249143, 12194, 11300, 16617, 574, 201, 6, 185279, 1496, 23150,
1328, 24593, 30993, 2117, 75032, 2903, 5033, 10222, 230286, 5378, 468, 7, 3038,
21135, 4324, 7807, 23694, 19740, 8412, 65243, 16280, 14133, 43735, 4763, 249143,
2], [0, 5675, 3679, 7102, 56460, 316, 13664, 39, 21260, 4450, 5259, 316, 348,
11, 2749, 56460, 316, 36, 5222, 124017, 117, 1946, 289, 10, 344, 32067, 50082,
203, 2465, 280, 1479, 7840, 144919, 72494, 203, 6229, 307, 47311, 229, 316,
2591, 1870, 153676, 153676, 1018, 34, 399, 91, 24948, 61957, 61957, 927, 2198,
2]]
```

[47]:
```
batch_ordered_labels[0]
```

[47]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

0.10.3 Padding

[48]:
```
py_inputs = []
py_attn_masks = []
py_labels = []

# For each batch...
for (batch_inputs, batch_labels) in zip(batch_ordered_sentences,␣
 ↪batch_ordered_labels):

    # New version of the batch, this time with padded sequences and now with
    # attention masks defined.
    batch_padded_inputs = []
    batch_attn_masks = []

    # First, find the longest sample in the batch.
    # Note that the sequences do currently include the special tokens!
    max_size = max([len(sen) for sen in batch_inputs])

    #print('Max size:', max_size)

    # For each input in this batch...
    for sen in batch_inputs:
```

```
        # How many pad tokens do we need to add?
        num_pads = max_size - len(sen)

        # Add `num_pads` padding tokens to the end of the sequence.
        padded_input = sen + [tokenizer.pad_token_id]*num_pads

        # Define the attention mask--it's just a `1` for every real token
        # and a `0` for every padding token.
        attn_mask = [1] * len(sen) + [0] * num_pads

        # Add the padded results to the batch.
        batch_padded_inputs.append(padded_input)
        batch_attn_masks.append(attn_mask)

    # Our batch has been padded, so we need to save this updated batch.
    # We also need the inputs to be PyTorch tensors, so we'll do that here.
    py_inputs.append(torch.tensor(batch_padded_inputs))
    py_attn_masks.append(torch.tensor(batch_attn_masks))
    py_labels.append(torch.tensor(batch_labels))
```

0.10.4 Check the number of token reductions because of smart batching

[49]:
```
# Get the new list of lengths after sorting.

padded_lengths = []

# For each batch...
for batch in py_inputs:

    # For each sample...
    for s in batch:

        # Record its length.
        padded_lengths.append(len(s))

# Sum up the lengths to the get the total number of tokens after smart batching.
smart_token_count = np.sum(padded_lengths)

# To get the total number of tokens in the dataset using fixed padding, it's
# as simple as the number of samples times our `max_len` parameter (that we
# would pad everything to).
fixed_token_count = len(train_sentences) * max_input_length

# Calculate the percentage reduction.
prcnt_reduced = (fixed_token_count - smart_token_count) /␣
 ↪float(fixed_token_count)
```

```
print('Total tokens:')
print('   Fixed Padding: {:,}'.format(fixed_token_count))
print('  Smart Batching: {:,}  ({:.1%} less)'.format(smart_token_count,␣
↪prcnt_reduced))
```

```
Total tokens:
   Fixed Padding: 4,321,600
  Smart Batching: 264,648  (93.9% less)
```

0.11 Load the model configuration from the transformers library using AutoConfig

[50]:
```
# Load the Config object, with an output configured for classification.
config = AutoConfig.from_pretrained(pretrained_model_name_or_path=model_name,
                                    num_labels=5)

print('Config type:', str(type(config)), '\n')
```

```
Config type: <class
'transformers.models.xlm_roberta.configuration_xlm_roberta.XLMRobertaConfig'>
```

[51]:
```
model = AutoModelForSequenceClassification.
↪from_pretrained(pretrained_model_name_or_path=model_name,config = config)
```

```
model.safetensors:   0%|          | 0.00/1.12G [00:00<?, ?B/s]
```

```
Some weights of XLMRobertaForSequenceClassification were not initialized from
the model checkpoint at xlm-roberta-base and are newly initialized:
['classifier.dense.bias', 'classifier.dense.weight', 'classifier.out_proj.bias',
'classifier.out_proj.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

[52]:
```
model.cuda()
```

[52]:
```
XLMRobertaForSequenceClassification(
  (roberta): XLMRobertaModel(
    (embeddings): XLMRobertaEmbeddings(
      (word_embeddings): Embedding(250002, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): XLMRobertaEncoder(
      (layer): ModuleList(
```

```
        (0-11): 12 x XLMRobertaLayer(
          (attention): XLMRobertaAttention(
            (self): XLMRobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): XLMRobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): XLMRobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): XLMRobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (classifier): XLMRobertaClassificationHead(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (out_proj): Linear(in_features=768, out_features=5, bias=True)
  )
)
```

0.12 Load the model from the transformers library using AutoModel

[53]:
```
# # Load the pre-trained model for classification, passing in the `config` from
# # above.
# bert = AutoModel.from_pretrained(
#     pretrained_model_name_or_path=model_name,
#     config=config)

# print('\nModel type:', str(type(bert)))
```

[54]:
```
# # freeze all the parameters
# for param in bert.parameters():
#     param.requires_grad = False
```

0.13 Creating a custom BERT model

[55]:
```
# class BERT_Model(nn.Module):
#     ## defining the constructor for the class
#     def __init__(self, bert,num_labels):
#       ## calling the super class constructor
#       super(BERT_Model, self).__init__()
#       ## adding the bert layer to the model
#       self.bert = bert
#       # relu activation function
#       self.relu =  nn.ReLU()
#       # adding a dense layer to our custom model
#       self.fc1 = nn.Linear(768,512)
#       # adding another dense layer to our custom model ,i.e., the Output layer
#       self.fc2 = nn.Linear(512,num_labels)
#       # adding a softmax  activation function for our custom model's output␣
↪layer
#       self.softmax = nn.LogSoftmax(dim=1)

#     #defining the forward pass
#     def forward(self, input_id, mask):
#       #pass the inputs to the model
#       outputs = self.bert(input_id,mask)

#       last_hidden_state = outputs.last_hidden_state       ## last hidden␣
↪state from the model
#       pooler_output = outputs.pooler_output               ## pooler output␣
↪from the model
#       ## adding a fully connected layer to the BERT model
#       x = self.fc1(pooler_output)
#       ## applying relu activation function
#       x = self.relu(x)
#       # the final output layer
#       x = self.fc2(x)
#       # apply softmax activation to our output layer
#       x = self.softmax(x)

#       return x
```

[56]:
```
# print('\nLoading model ...')
# # pass the pre-trained BERT to our define architecture
# model = BERT_Model(bert,num_labels=3)

# model.cuda()
```

0.14 Custom Loss function

[57]:
```
# # convert class weights to tensor
# weights= torch.tensor(class_wts,dtype=torch.float)
# weights = weights.to(device)

# # loss function
# cross_entropy  = nn.NLLLoss(weight=weights)
```

0.15 Loading Optimizer

[58]:
```
# Note: AdamW is a class from the huggingface library (as opposed to pytorch)
# I believe the 'W' stands for 'Weight Decay fix"
optimizer = AdamW(model.parameters(),
                  lr = 2e-5, # This is the value Michael used.
                  eps = 1e-8 # args.adam_epsilon  - default is 1e-8.
                )
```

```
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:429:
FutureWarning: This implementation of AdamW is deprecated and will be removed in
a future version. Use the PyTorch implementation torch.optim.AdamW instead, or
set `no_deprecation_warning=True` to disable this warning
  warnings.warn(
```

0.16 Loading lr scheduler

[59]:
```
# Number of training epochs. I chose to train for 1 simply because the training
# time is long. More epochs may improve the model's accuracy.
epochs = 4

# Total number of training steps is [number of batches] x [number of epochs].
# Note that it's the number of *batches*, not *samples*!
total_steps = len(py_inputs) * epochs

# Create the learning rate scheduler.
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0, # Default␣
 ↪value in run_glue.py
                                            num_training_steps = total_steps)
```

0.17 Training Loop

[60]:
```
# We'll store a number of quantities such as training and validation loss,
# validation accuracy, and timings.
training_stats = {
    'epoch':[],
    'train_loss':[],
```

```
    'Training Time':[],
    'val_loss':[],
    'Validation Time':[],
    'train_acc':[],
    'val_acc':[]
}

# Update every `update_interval` batches.
update_interval = good_update_interval(total_iters=len(py_inputs),␣
 ↪num_desired_updates=10)

# Measure the total training time for the whole run.
total_t0 = time.time()

# For each epoch...
for epoch_i in range(0, epochs):

    predictions = []
    true_labels = []

    # ========================================
    #               Training
    # ========================================

    # Perform one full pass over the training set.

    print("")
    print('======== Epoch {:} / {:} ========'.format(epoch_i + 1, epochs))

    # At the start of each epoch (except for the first) we need to re-randomize
    # our training data.
    if epoch_i > 0:
        # Use our `make_smart_batches` function (from 6.1.) to re-shuffle the
        # dataset into new batches.
        (py_inputs, py_attn_masks, py_labels) =␣
 ↪make_smart_batches(train_sentences, train_labels,␣
 ↪batch_size,tokenizer,max_input_length)

    print('Training on {:,} batches...'.format(len(py_inputs)))

    # Measure how long the training epoch takes.
    t0 = time.time()

    # Reset the total loss for this epoch.
    total_train_loss = 0

    # Put the model into training mode. Don't be mislead--the call to
```

```
    # `train` just changes the *mode*, it doesn't *perform* the training.
    # `dropout` and `batchnorm` layers behave differently during training
    # vs. test (source: https://stackoverflow.com/questions/51433378/
↪what-does-model-train-do-in-pytorch)
    model.train()

    # For each batch of training data...
    for step in range(0, len(py_inputs)):

        # Progress update every, e.g., 100 batches.
        if step % update_interval == 0 and not step == 0:
            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)

            # Calculate the time remaining based on our progress.
            steps_per_sec = (time.time() - t0) / step
            remaining_sec = steps_per_sec * (len(py_inputs) - step)
            remaining = format_time(remaining_sec)

            # Report progress.
            print('  Batch {:>7,}  of  {:>7,}.    Elapsed: {:}.  Remaining: {:
↪}'.format(step, len(py_inputs), elapsed, remaining))

        # Copy the current training batch to the GPU using the `to` method.
        b_input_ids = py_inputs[step].to(device)
        b_input_mask = py_attn_masks[step].to(device)
        b_labels = py_labels[step].to(device)

        # Always clear any previously calculated gradients before performing a
        # backward pass.
        model.zero_grad()

        # Perform a forward pass (evaluate the model on this training batch).
        # The call returns the loss (because we provided labels) and the
        # "logits"--the model outputs prior to activation.
        output = model(b_input_ids,
                             token_type_ids=None,
                             attention_mask=b_input_mask,
                             labels=b_labels)

        # Accumulate the training loss over all of the batches so that we can
        # calculate the average loss at the end. `loss` is a Tensor containing a
        # single value; the `.item()` function just returns the Python value
        # from the tensor.
        loss = output.loss
        logits = output.logits
```

```
        # Move logits and labels to CPU
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()

        # Store predictions and true labels
        predictions.append(logits)
        true_labels.append(label_ids)

        # Accumulate the training loss over all of the batches so that we can
        # calculate the average loss at the end. `loss` is a Tensor containing a
        # single value; the `.item()` function just returns the Python value
        # from the tensor.
        total_train_loss += loss.item()

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

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

        # Update parameters and take a step using the computed gradient.
        # The optimizer dictates the "update rule"--how the parameters are
        # modified based on their gradients, the learning rate, etc.
        optimizer.step()

        # Update the learning rate.
        scheduler.step()

    # Calculate the average loss over all of the batches.
    avg_train_loss = total_train_loss / len(py_inputs)

    training_accuracy = check_accuracy(predictions,true_labels)

    # Measure how long this epoch took.
    training_time = format_time(time.time() - t0)

    print("")
    print("  Average training loss: {0:.2f}".format(avg_train_loss))
    print("  Training Accuracy: {0:.2f}".format(training_accuracy))
    print("  Training epoch took: {:}".format(training_time))

    (py_inputs, py_attn_masks, py_labels) = make_smart_batches(val_sentences,␣
↪val_labels, batch_size ,tokenizer,max_input_length)
    val_loss,val_accuracy,validation_time = eval_model(model,py_inputs,␣
↪py_attn_masks, py_labels)
    # Record all statistics from this epoch.
```

```
    print("")
    print("  Average validation loss: {0:.2f}".format(val_loss))
    print("  Validation Accuracy: {0:.2f}".format(val_accuracy))
    print("  Validation epoch took: {:}".format(validation_time))


    training_stats['epoch'].append(epoch_i + 1)
    training_stats['train_loss'].append(avg_train_loss)
    training_stats['Training Time'].append(training_time)
    training_stats['val_loss'].append(val_loss)
    training_stats['Validation Time'].append(validation_time)
    training_stats['train_acc'].append(training_accuracy)
    training_stats['val_acc'].append(val_accuracy)

print("")
print("Training complete!")

print("Total training took {:} (h:mm:ss)".format(format_time(time.
 ↪time()-total_t0)))
```

```
======== Epoch 1 / 4 ========
Training on 676 batches…
  Batch      70  of      676.    Elapsed: 0:00:12.  Remaining: 0:01:44
  Batch     140  of      676.    Elapsed: 0:00:23.  Remaining: 0:01:29
  Batch     210  of      676.    Elapsed: 0:00:34.  Remaining: 0:01:15
  Batch     280  of      676.    Elapsed: 0:00:46.  Remaining: 0:01:05
  Batch     350  of      676.    Elapsed: 0:00:57.  Remaining: 0:00:53
  Batch     420  of      676.    Elapsed: 0:01:08.  Remaining: 0:00:41
  Batch     490  of      676.    Elapsed: 0:01:19.  Remaining: 0:00:30
  Batch     560  of      676.    Elapsed: 0:01:30.  Remaining: 0:00:19
  Batch     630  of      676.    Elapsed: 0:01:42.  Remaining: 0:00:07

  Average training loss: 0.24
  Training Accuracy: 0.96
  Training epoch took: 0:01:50
Creating Smart Batches from 1,766 examples with batch size 16…

Tokenizing 1,766 samples…
  Tokenized 0 samples.
  Tokenized 200 samples.
  Tokenized 400 samples.
  Tokenized 600 samples.
  Tokenized 800 samples.
  Tokenized 1,000 samples.
  Tokenized 1,200 samples.
```

```
  Tokenized 1,400 samples.
  Tokenized 1,600 samples.
DONE.
     1,766 samples

     1,766 samples after sorting

Creating batches of size 16…

  DONE - Selected 111 batches.

Padding out sequences within each batch…
  DONE.
Doing validation on 111 sentences…
  Batch      10  of      111.    Elapsed: 0:00:00.  Remaining: 0:00:02
  Batch      20  of      111.    Elapsed: 0:00:00.  Remaining: 0:00:02
  Batch      30  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch      40  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch      50  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch      60  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch      70  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch      80  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch      90  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:00
  Batch     100  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:00
  Batch     110  of      111.    Elapsed: 0:00:03.  Remaining: 0:00:00

  Average validation loss: 0.19
  Validation Accuracy: 0.97
  Validation epoch took: 0:00:03

======== Epoch 2 / 4 ========
Creating Smart Batches from 10,804 examples with batch size 16…

Tokenizing 10,804 samples…
  Tokenized 0 samples.
  Tokenized 1,000 samples.
  Tokenized 2,000 samples.
  Tokenized 3,000 samples.
  Tokenized 4,000 samples.
  Tokenized 5,000 samples.
  Tokenized 6,000 samples.
  Tokenized 7,000 samples.
  Tokenized 8,000 samples.
  Tokenized 9,000 samples.
  Tokenized 10,000 samples.
DONE.
    10,804 samples
```

```
    10,804 samples after sorting

Creating batches of size 16…

  DONE - Selected 676 batches.

Padding out sequences within each batch…
  DONE.
Training on 676 batches…
  Batch      70  of      676.    Elapsed: 0:00:11.  Remaining: 0:01:35
  Batch     140  of      676.    Elapsed: 0:00:22.  Remaining: 0:01:25
  Batch     210  of      676.    Elapsed: 0:00:33.  Remaining: 0:01:14
  Batch     280  of      676.    Elapsed: 0:00:45.  Remaining: 0:01:03
  Batch     350  of      676.    Elapsed: 0:00:56.  Remaining: 0:00:52
  Batch     420  of      676.    Elapsed: 0:01:07.  Remaining: 0:00:41
  Batch     490  of      676.    Elapsed: 0:01:18.  Remaining: 0:00:30
  Batch     560  of      676.    Elapsed: 0:01:30.  Remaining: 0:00:19
  Batch     630  of      676.    Elapsed: 0:01:41.  Remaining: 0:00:07

  Average training loss: 0.21
  Training Accuracy: 0.96
  Training epoch took: 0:01:49
Creating Smart Batches from 1,766 examples with batch size 16…

Tokenizing 1,766 samples…
  Tokenized 0 samples.
  Tokenized 200 samples.
  Tokenized 400 samples.
  Tokenized 600 samples.
  Tokenized 800 samples.
  Tokenized 1,000 samples.
  Tokenized 1,200 samples.
  Tokenized 1,400 samples.
  Tokenized 1,600 samples.
DONE.
     1,766 samples

     1,766 samples after sorting

Creating batches of size 16…

  DONE - Selected 111 batches.

Padding out sequences within each batch…
  DONE.
Doing validation on 111 sentences…
  Batch      10  of      111.    Elapsed: 0:00:00.  Remaining: 0:00:02
  Batch      20  of      111.    Elapsed: 0:00:00.  Remaining: 0:00:02
```

```
  Batch      30  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch      40  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch      50  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch      60  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch      70  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch      80  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch      90  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch     100  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:00
  Batch     110  of      111.    Elapsed: 0:00:03.  Remaining: 0:00:00

  Average validation loss: 0.17
  Validation Accuracy: 0.97
  Validation epoch took: 0:00:03

======== Epoch 3 / 4 ========
Creating Smart Batches from 10,804 examples with batch size 16…

Tokenizing 10,804 samples…
  Tokenized 0 samples.
  Tokenized 1,000 samples.
  Tokenized 2,000 samples.
  Tokenized 3,000 samples.
  Tokenized 4,000 samples.
  Tokenized 5,000 samples.
  Tokenized 6,000 samples.
  Tokenized 7,000 samples.
  Tokenized 8,000 samples.
  Tokenized 9,000 samples.
  Tokenized 10,000 samples.
DONE.
    10,804 samples

    10,804 samples after sorting

Creating batches of size 16…

  DONE - Selected 676 batches.

Padding out sequences within each batch…
  DONE.
Training on 676 batches…
  Batch      70  of      676.    Elapsed: 0:00:11.  Remaining: 0:01:35
  Batch     140  of      676.    Elapsed: 0:00:22.  Remaining: 0:01:25
  Batch     210  of      676.    Elapsed: 0:00:35.  Remaining: 0:01:17
  Batch     280  of      676.    Elapsed: 0:00:47.  Remaining: 0:01:06
  Batch     350  of      676.    Elapsed: 0:00:58.  Remaining: 0:00:54
  Batch     420  of      676.    Elapsed: 0:01:09.  Remaining: 0:00:42
  Batch     490  of      676.    Elapsed: 0:01:21.  Remaining: 0:00:31
```

```
  Batch     560  of      676.    Elapsed: 0:01:32.  Remaining: 0:00:19
  Batch     630  of      676.    Elapsed: 0:01:43.  Remaining: 0:00:08

  Average training loss: 0.19
  Training Accuracy: 0.96
  Training epoch took: 0:01:51
Creating Smart Batches from 1,766 examples with batch size 16…

Tokenizing 1,766 samples…
  Tokenized 0 samples.
  Tokenized 200 samples.
  Tokenized 400 samples.
  Tokenized 600 samples.
  Tokenized 800 samples.
  Tokenized 1,000 samples.
  Tokenized 1,200 samples.
  Tokenized 1,400 samples.
  Tokenized 1,600 samples.
DONE.
     1,766 samples

     1,766 samples after sorting

Creating batches of size 16…

  DONE - Selected 111 batches.

Padding out sequences within each batch…
  DONE.
Doing validation on 111 sentences…
  Batch      10  of      111.    Elapsed: 0:00:00.  Remaining: 0:00:02
  Batch      20  of      111.    Elapsed: 0:00:00.  Remaining: 0:00:02
  Batch      30  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch      40  of      111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch      50  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:02
  Batch      60  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:02
  Batch      70  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch      80  of      111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch      90  of      111.    Elapsed: 0:00:03.  Remaining: 0:00:01
  Batch     100  of      111.    Elapsed: 0:00:03.  Remaining: 0:00:00
  Batch     110  of      111.    Elapsed: 0:00:03.  Remaining: 0:00:00

  Average validation loss: 0.17
  Validation Accuracy: 0.96
  Validation epoch took: 0:00:03

======== Epoch 4 / 4 ========
Creating Smart Batches from 10,804 examples with batch size 16…
```

```
Tokenizing 10,804 samples…
  Tokenized 0 samples.
  Tokenized 1,000 samples.
  Tokenized 2,000 samples.
  Tokenized 3,000 samples.
  Tokenized 4,000 samples.
  Tokenized 5,000 samples.
  Tokenized 6,000 samples.
  Tokenized 7,000 samples.
  Tokenized 8,000 samples.
  Tokenized 9,000 samples.
  Tokenized 10,000 samples.
DONE.
    10,804 samples

    10,804 samples after sorting

Creating batches of size 16…

  DONE - Selected 676 batches.

Padding out sequences within each batch…
  DONE.
Training on 676 batches…
  Batch      70  of      676.    Elapsed: 0:00:11.  Remaining: 0:01:36
  Batch     140  of      676.    Elapsed: 0:00:22.  Remaining: 0:01:25
  Batch     210  of      676.    Elapsed: 0:00:33.  Remaining: 0:01:14
  Batch     280  of      676.    Elapsed: 0:00:44.  Remaining: 0:01:03
  Batch     350  of      676.    Elapsed: 0:00:55.  Remaining: 0:00:52
  Batch     420  of      676.    Elapsed: 0:01:07.  Remaining: 0:00:41
  Batch     490  of      676.    Elapsed: 0:01:18.  Remaining: 0:00:30
  Batch     560  of      676.    Elapsed: 0:01:30.  Remaining: 0:00:19
  Batch     630  of      676.    Elapsed: 0:01:42.  Remaining: 0:00:07

  Average training loss: 0.17
  Training Accuracy: 0.96
  Training epoch took: 0:01:50
Creating Smart Batches from 1,766 examples with batch size 16…

Tokenizing 1,766 samples…
  Tokenized 0 samples.
  Tokenized 200 samples.
  Tokenized 400 samples.
  Tokenized 600 samples.
  Tokenized 800 samples.
  Tokenized 1,000 samples.
  Tokenized 1,200 samples.
```

```
  Tokenized 1,400 samples.
  Tokenized 1,600 samples.
DONE.
     1,766 samples

     1,766 samples after sorting

Creating batches of size 16…

  DONE - Selected 111 batches.

Padding out sequences within each batch…
  DONE.
Doing validation on 111 sentences…
  Batch     10  of     111.    Elapsed: 0:00:00.  Remaining: 0:00:02
  Batch     20  of     111.    Elapsed: 0:00:00.  Remaining: 0:00:02
  Batch     30  of     111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch     40  of     111.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch     50  of     111.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch     60  of     111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch     70  of     111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch     80  of     111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch     90  of     111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch    100  of     111.    Elapsed: 0:00:03.  Remaining: 0:00:00
  Batch    110  of     111.    Elapsed: 0:00:03.  Remaining: 0:00:00

  Average validation loss: 0.17
  Validation Accuracy: 0.96
  Validation epoch took: 0:00:03

Training complete!
Total training took 0:07:39 (h:mm:ss)
```

[61]:
```
plot_training_history(training_stats)
```

0.18 Evaluating Performance Over Training Set

[62]:
```
(py_inputs, py_attn_masks, py_labels) = make_smart_batches(train_sentences,␣
↪train_labels, batch_size ,tokenizer,max_input_length)

y_pred , y_true = get_predictions(py_inputs, py_attn_masks, py_labels)
```

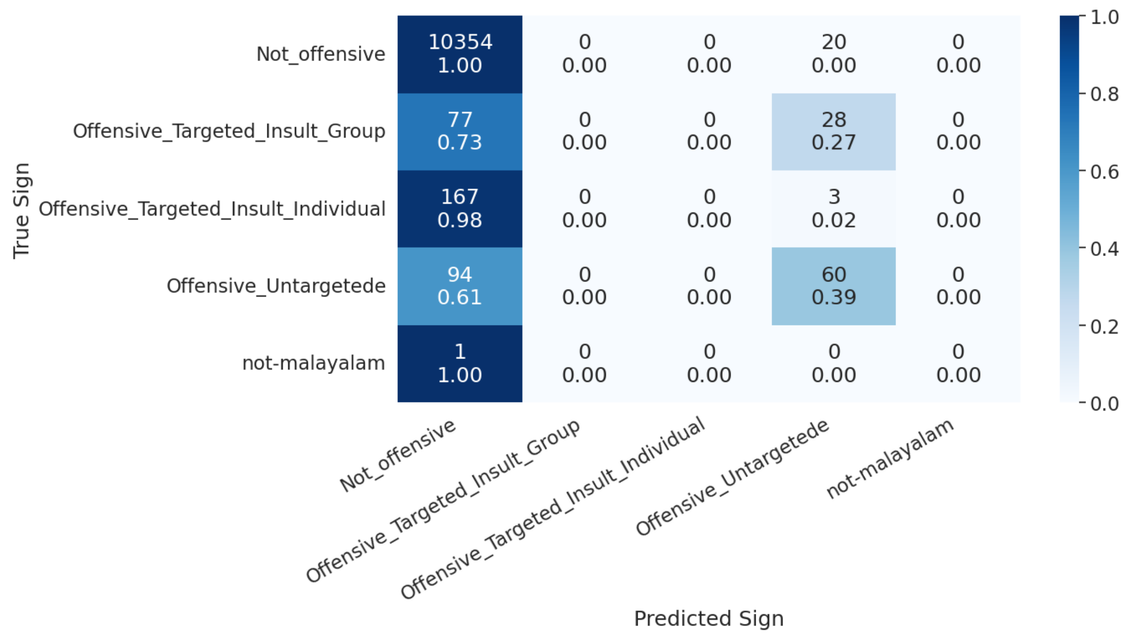
```
Creating Smart Batches from 10,804 examples with batch size 16…

Tokenizing 10,804 samples…
  Tokenized 0 samples.
  Tokenized 1,000 samples.
  Tokenized 2,000 samples.
  Tokenized 3,000 samples.
  Tokenized 4,000 samples.
  Tokenized 5,000 samples.
  Tokenized 6,000 samples.
  Tokenized 7,000 samples.
  Tokenized 8,000 samples.
  Tokenized 9,000 samples.
  Tokenized 10,000 samples.
DONE.
    10,804 samples

    10,804 samples after sorting

Creating batches of size 16…

  DONE - Selected 676 batches.

Padding out sequences within each batch…
  DONE.
Predicting labels for 676 test batches…
  Batch      70  of      676.    Elapsed: 0:00:02.  Remaining: 0:00:14
  Batch     140  of      676.    Elapsed: 0:00:03.  Remaining: 0:00:13
  Batch     210  of      676.    Elapsed: 0:00:05.  Remaining: 0:00:11
  Batch     280  of      676.    Elapsed: 0:00:07.  Remaining: 0:00:10
  Batch     350  of      676.    Elapsed: 0:00:08.  Remaining: 0:00:08
  Batch     420  of      676.    Elapsed: 0:00:10.  Remaining: 0:00:06
  Batch     490  of      676.    Elapsed: 0:00:12.  Remaining: 0:00:04
  Batch     560  of      676.    Elapsed: 0:00:13.  Remaining: 0:00:03
  Batch     630  of      676.    Elapsed: 0:00:15.  Remaining: 0:00:01
```

[63]:
```
print(classification_report(y_true, y_pred, target_names=class_names))
```

```
                                      precision    recall  f1-score   support

                       Not_offensive       0.97      1.00      0.98     10374
      Offensive_Targeted_Insult_Group       0.00      0.00      0.00       105
Offensive_Targeted_Insult_Individual       0.00      0.00      0.00       170
                 Offensive_Untargetede       0.54      0.39      0.45       154
                       not-malayalam       0.00      0.00      0.00         1

                            accuracy                           0.96     10804
                           macro avg       0.30      0.28      0.29     10804
                        weighted avg       0.94      0.96      0.95     10804

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[64]:

```
cm = confusion_matrix(y_true, y_pred)
show_confusion_matrix(cm, class_names)
```



0.19 Evaluating Performance Over Validation Set

[65]:
```
(py_inputs, py_attn_masks, py_labels) = make_smart_batches(val_sentences,␣
↪val_labels, batch_size ,tokenizer,max_input_length)

y_pred , y_true = get_predictions(py_inputs, py_attn_masks, py_labels)
```

```
Creating Smart Batches from 1,766 examples with batch size 16…

Tokenizing 1,766 samples…
  Tokenized 0 samples.
  Tokenized 200 samples.
  Tokenized 400 samples.
  Tokenized 600 samples.
  Tokenized 800 samples.
  Tokenized 1,000 samples.
  Tokenized 1,200 samples.
  Tokenized 1,400 samples.
  Tokenized 1,600 samples.
DONE.
     1,766 samples

     1,766 samples after sorting

Creating batches of size 16…
```

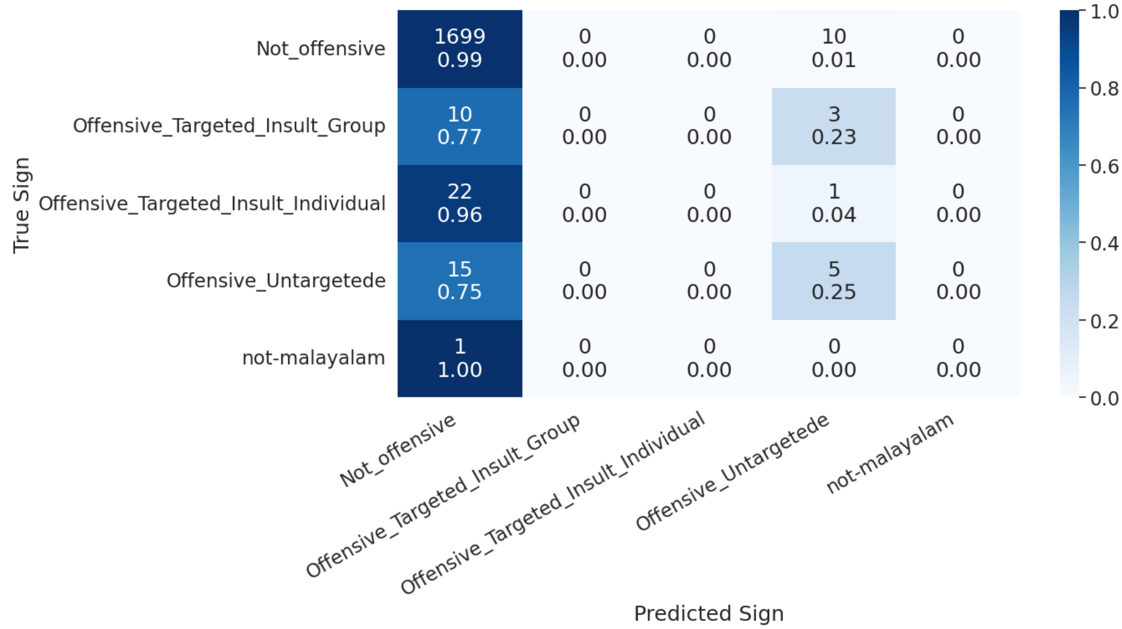
```
  DONE - Selected 111 batches.

Padding out sequences within each batch…
  DONE.
Predicting labels for 111 test batches…
  Batch     10  of     111.    Elapsed: 0:00:00.  Remaining: 0:00:03
  Batch     20  of     111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch     30  of     111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch     40  of     111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch     50  of     111.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch     60  of     111.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch     70  of     111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch     80  of     111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch     90  of     111.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch    100  of     111.    Elapsed: 0:00:02.  Remaining: 0:00:00
  Batch    110  of     111.    Elapsed: 0:00:03.  Remaining: 0:00:00
```

[66]:

```
print(classification_report(y_true, y_pred, target_names=class_names))
```

```
                                    precision    recall  f1-score   support

                      Not_offensive       0.97      0.99      0.98      1709
    Offensive_Targeted_Insult_Group       0.00      0.00      0.00        13
Offensive_Targeted_Insult_Individual       0.00      0.00      0.00        23
               Offensive_Untargetede       0.26      0.25      0.26        20
                      not-malayalam       0.00      0.00      0.00         1

                           accuracy                           0.96      1766
                          macro avg       0.25      0.25      0.25      1766
                       weighted avg       0.94      0.96      0.95      1766


/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[67]:
```
cm = confusion_matrix(y_true, y_pred)
show_confusion_matrix(cm, class_names)
```



0.20 Saving the model

[68]:
```
print(model)
```

```
XLMRobertaForSequenceClassification(
  (roberta): XLMRobertaModel(
    (embeddings): XLMRobertaEmbeddings(
      (word_embeddings): Embedding(250002, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): XLMRobertaEncoder(
      (layer): ModuleList(
        (0-11): 12 x XLMRobertaLayer(
          (attention): XLMRobertaAttention(
            (self): XLMRobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
```

```
            (output): XLMRobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): XLMRobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): XLMRobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (classifier): XLMRobertaClassificationHead(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (out_proj): Linear(in_features=768, out_features=5, bias=True)
  )
)
```

[70]:
```
torch.save(model,"/content/drive/MyDrive/ChallengingExpt/XLMR_Malayalam_v1")
```

0.21 Loading the model

[72]:
```
model = torch.load('/content/drive/MyDrive/ChallengingExpt/
↪XLMR_Malayalam_v1',map_location=device)
```

[73]:
```
model.cuda()
```

[73]:
```
XLMRobertaForSequenceClassification(
  (roberta): XLMRobertaModel(
    (embeddings): XLMRobertaEmbeddings(
      (word_embeddings): Embedding(250002, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): XLMRobertaEncoder(
      (layer): ModuleList(
        (0-11): 12 x XLMRobertaLayer(
```

```
          (attention): XLMRobertaAttention(
            (self): XLMRobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): XLMRobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): XLMRobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): XLMRobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (classifier): XLMRobertaClassificationHead(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (out_proj): Linear(in_features=768, out_features=5, bias=True)
  )
)
```

0.22 Making Predictions on Test Set

[74]:
```python
test_df_path = join(dataset_dir,'Malayalam_test_1-5_cleaned.csv')
test_df = pd.read_csv(test_df_path, delimiter='\t', header=None, names=['text',
 ↪'label'])
```

[75]:
```python
test_df
```

[75]:
```
                                                    text        label
0                                                   text     category
1                      Suraj       …  Not_offensive
2                          vivo vid…  Not_offensive
3            Fefka ee padam release cheyyan samadhicho?  Not_offensive
4        ..                       …  Not_offensive
```

```
…                                                   …             …
1785                                Not_offensive
1786  Swargatthil ninnu purathaakkappetta daivatthin…  Not_offensive
1787  Ivide Palakkad Jayettan Fans club nnu ashamsak…  Not_offensive
1788                                Not_offensive
1789                             …  Not_offensive

[1790 rows x 2 columns]
```

0.23 Loading test_sentences

[76]:
```
test_sentences = test_df['text'].values
test_ids = test_df.index.values
```

[77]:
```
test_ids
```

[77]:
```
array([   0,    1,    2, …, 1787, 1788, 1789])
```

[78]:
```
test_sentences
```

[78]:
```
array(['text',
       '                 Suraj           ',
       '                     vivo videoorderil     ',
       …,
       'Ivide Palakkad Jayettan Fans club nnu ashamsakal!!! Adichal romam
ennikum athaanu Thrissur pooram',
       '                     ',
       '                         Sep6 appo pwlikale'],
      dtype=object)
```

[79]:
```
(py_inputs, py_attn_masks,py_ids) =␣
 ↪make_smart_batches_on_test(test_sentences,test_ids,␣
 ↪16,tokenizer,max_input_length)
```

```
Creating Smart Batches from 1,790 examples with batch size 16…

Tokenizing 1,790 samples…
  Tokenized 0 samples.
  Tokenized 200 samples.
  Tokenized 400 samples.
  Tokenized 600 samples.
  Tokenized 800 samples.
  Tokenized 1,000 samples.
  Tokenized 1,200 samples.
  Tokenized 1,400 samples.
  Tokenized 1,600 samples.
DONE.
```

```
    1,790 samples

    1,790 samples after sorting

Creating batches of size 16…

  DONE - Selected 112 batches.

Padding out sequences within each batch…
  DONE.
```

0.24 Evaluating over test set

[80]:
```
y_ids,y_preds  = get_predictions_test(py_inputs, py_attn_masks,py_ids)
```

```
Predicting labels for 112 test batches…
  Batch    10  of    112.    Elapsed: 0:00:00.  Remaining: 0:00:03
  Batch    20  of    112.    Elapsed: 0:00:01.  Remaining: 0:00:03
  Batch    30  of    112.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch    40  of    112.    Elapsed: 0:00:01.  Remaining: 0:00:02
  Batch    50  of    112.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch    60  of    112.    Elapsed: 0:00:01.  Remaining: 0:00:01
  Batch    70  of    112.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch    80  of    112.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch    90  of    112.    Elapsed: 0:00:02.  Remaining: 0:00:01
  Batch   100  of    112.    Elapsed: 0:00:02.  Remaining: 0:00:00
  Batch   110  of    112.    Elapsed: 0:00:03.  Remaining: 0:00:00
```

[87]:
```
print(classification_report(y_true, y_pred, target_names=class_names))
```

```
                                     precision    recall  f1-score   support

                       Not_offensive       0.97      0.99      0.98      1709
      Offensive_Targeted_Insult_Group       0.00      0.00      0.00        13
Offensive_Targeted_Insult_Individual       0.00      0.00      0.00        23
                Offensive_Untargetede       0.26      0.25      0.26        20
                        not-malayalam       0.00      0.00      0.00         1

                             accuracy                           0.96      1766
                            macro avg       0.25      0.25      0.25      1766
                         weighted avg       0.94      0.96      0.95      1766


/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[88]:

```python
import csv

# Assuming you have obtained y_ids, y_preds, and test_sentences
# y_ids: List of text IDs
# y_preds: List of predicted labels
# test_sentences: List of test sentences

# Define the file path for the CSV file
csv_file_path = "/content/drive/MyDrive/ChallengingExpt/prediction.csv"

# Write the data into the CSV file
with open(csv_file_path, "w", newline="", encoding="utf-8") as csv_file:
    csv_writer = csv.writer(csv_file)
    # Write the header row
    csv_writer.writerow(["Text id", "Test sentences", "Predicted label"])
    # Write each row containing text ID, test sentence, and predicted label
    for text_id, test_sentence, predicted_label in zip(y_ids, test_sentences,␣
 ↪y_preds):
        csv_writer.writerow([text_id, test_sentence, predicted_label])

print(f"Prediction data has been written to {csv_file_path}")
```

```
Prediction data has been written to
/content/drive/MyDrive/ChallengingExpt/prediction.csv
```

[81]:

```python
print(y_preds)
```

```
[0 0 0 … 0 0 0]
```

[82]:

```python
print(y_ids)
```

```
[ 144  151  173 … 1469 1080  477]
```

[83]:

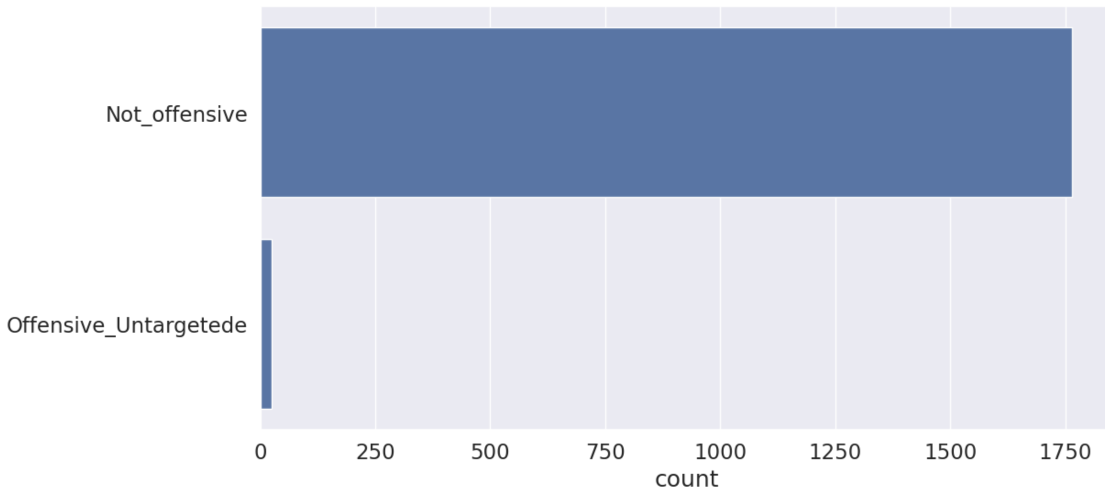
```python
le.inverse_transform(y_preds)
```

[83]:

```
array(['Not_offensive', 'Not_offensive', 'Not_offensive', …,
       'Not_offensive', 'Not_offensive', 'Not_offensive'], dtype=object)
```

[84]:
```
sns.countplot(y =le.inverse_transform(y_preds))
```

[84]: <Axes: xlabel='count'>



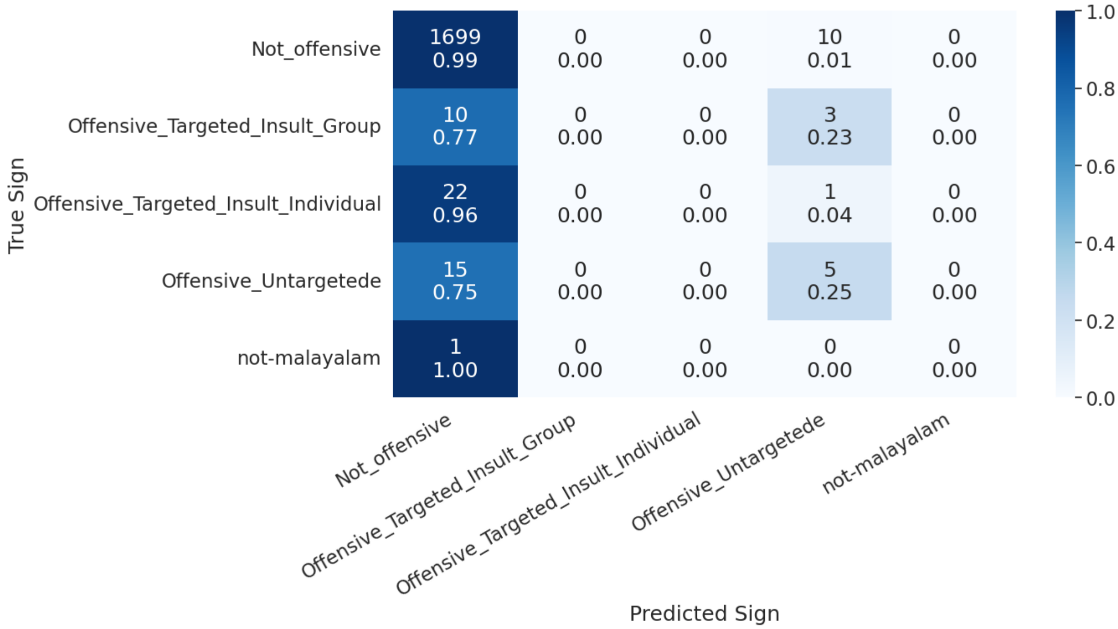
[85]:
```
len(y_ids)
```

[85]: 1790

[86]:
```
len(le.inverse_transform(y_preds))
```

[86]: 1790

[89]:
```
cm = confusion_matrix(y_true, y_pred)
show_confusion_matrix(cm, class_names)
```


True Sign
Not_offensive
Offensive_Targeted_Insult_Group
Offensive_Targeted_Insult_Individual
Offensive_Untargetede
not-malayalam
1699
0.99
0
0.00
0
0.00
10
0.01
0
0.00
10
0.77
0
0.00
0
0.00
3
0.23
0
0.00
22
0.96
0
0.00
0
0.00
1
0.04
0
0.00
15
0.75
0
0.00
0
0.00
5
0.25
0
0.00
1
1.00
0
0.00
0
0.00
0
0.00
0
0.00
1.0
0.8
0.6
0.4
0.2
0.0
Not_offensive
Offensive_Targeted_Insult_Group
Offensive_Targeted_Insult_Individual
Offensive_Untargetede
not-malayalam
Predicted Sign

```
[ ]:
```