NATURAL LANGUAGE PROCESSING

Leveraging Machine learning for Low resource languages In Africa

DSA 2023 Summer School

University of Rwanda

Kigali – Rwanda

Dr. Andrew Kipkebut

INTRODUCTION

Natural Language processing is a subfield of artificial intelligence that focuses on enabling computers to understand, interpret and generate human language. This paper will focus on training and fine-tuning models for machine translation in low level languages.

Resources

- Download notebook from github.com/kigenchesire/English-to-Swahili-Translation-NLP-
- Download data from github.com/kigenchesire/English-to-Swahili-Translation-NLP-
- Read more on huggingface transformers on huggingface.co/docs/transformers/index
- More about University of Helsinki model blogs.helsinki.fi/language-technology/
- Attention is all you need Research paper arxiv.org/abs/1706.03762
- How to deploy Streamlit apps docs.streamlit.io/streamlit-community-cloud/get-started/share-your-app
- Environments and prework paper were shared earlier to your emails. Make follow the instructions on that paper

What is NLP

NLP stands for **Natural Language Processing**, which is a part of **Computer Science, Human language,** and **Artificial Intelligence**. It is the technology that is used by machines to understand, Analyse, manipulate, and interpret human's languages. It helps developers to organize knowledge for performing tasks such as **translation, automatic summarization, Named Entity Recognition (NER), speech recognition, relationship extraction,** and **topic segmentation**.

Applications of NLP include Spam detection, sentimental analysis, machine translation, speech detection, Chatbots etc.

Components of NLP

1. Natural Language Understanding (NLU)

Natural Language Understanding (NLU) helps the machine to understand and analyses human language by extracting the metadata from content such as concepts, entities, keywords, emotion, relations, and semantic roles.

2. Natural Language Generation (NLG) acts as a translator that converts the computerized data into natural language representation. It mainly involves Text planning, Sentence planning, and Text Realization.

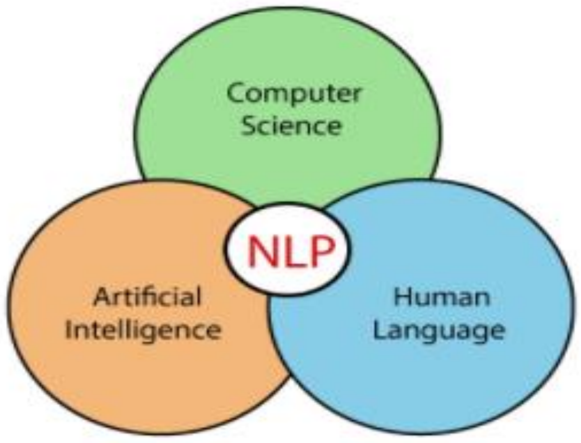


Figure 1 NLP

LOW RESOURCE LANGUAGES

Low-resource languages, also known as low-resourced or under-resourced languages, are languages with limited digital resources, including data, tools, and models, for NLP tasks. These languages are typically spoken by small populations or marginalized communities, and often lack the same level of linguistic and technological support as widely spoken languages. Examples of low-resource languages include many indigenous languages, regional dialects, and minority languages.

FINE TUNING ENGLISH-SWAHILI TRANSLATION MODEL

Fine tuning involves taking a pre-trained language model and adapting it to the specific task e.g. translation.

In this tutorial we will be fine tuning an English to Swahili translation model which was originally trained by University of Helsinki. This model has permissive open licenses which means you’re given permission to do all sorts of things with it (including commercial applications).

It will involve exposing the model to a large amount of bilingual data, consisting of pairs of English sentences and their corresponding Swahili translations. The model then learns to align the two languages and generate accurate translations.

Preparing the data

To fine-tune or train a translation model from scratch, we will need a dataset suitable for the task. We will use the already preprocessed English and Swahili pair sentences to translate from English to Swahili.

```
from datasets import load_dataset
```

```
dataset = load_dataset("csv", data_files="/kaggle/input/pretraindedsw/combined.csv"
```

```
Downloading and preparing dataset csv/default to /root/.cache/huggingface/dataset
s/csv/default-65bbae727179b21d/0.0.0/433e0ccc46f9880962cc2b12065189766fbb2bee57a22
1866138fb9203c83519...
Downloading data files:   0%|          | 0/1 [00:00<?, ?it/s]
Extracting data files:   0%|          | 0/1 [00:00<?, ?it/s]
Dataset csv downloaded and prepared to /root/.cache/huggingface/datasets/csv/defau
lt-65bbae727179b21d/0.0.0/433e0ccc46f9880962cc2b12065189766fbb2bee57a221866138fb92
03c83519. Subsequent calls will reuse this data.
  0%|          | 0/1 [00:00<?, ?it/s]
```

Let’s have a look at the dataset

```
# a view on the dataset object
dataset
```

```
DatasetDict({
    train: Dataset({
        features: ['English sentence', 'Swahii Translation'],
        num_rows: 8492
    })
})
```

We have 8493 pairs of sentences, but in one single split, so we will need to create our own validation set.

Split dataset for training and testing

We will use a train_test_split () method that can help us. We’ll 90% of data will be used for training and the remaining 10% for testing. We’ll provide a seed for reproducibility.

Now let’s take a look at one element of the dataset:

```
split_datasets = dataset["train"].train_test_split(train_size=0.9, seed=20)
split_datasets
```

```
DatasetDict({
    train: Dataset({
        features: ['English sentence', 'Swahii Translation'],
        num_rows: 7642
    })
    test: Dataset({
        features: ['English sentence', 'Swahii Translation'],
        num_rows: 850
    })
})
```

A peep on the data

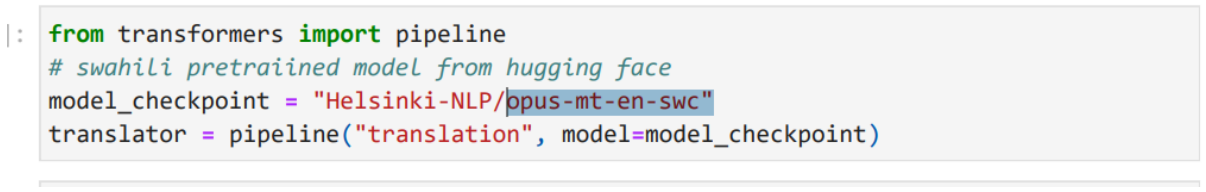
After splitting we get a dictionary with two sentences in the pair of languages we requested. Just to confirm the words were correctly aligned we pick one of the sentence pairs.

```
# a look on the dataset
split_datasets["train"][10]["English sentence"], split_datasets["train"][10]["Swahi
```

```
('We are going to build a new wall around the school.',
 'Tutaunda ukuta mpya kuzunguka shule.')
```

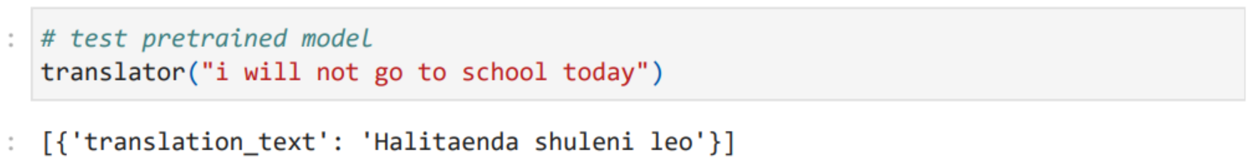
Load pretrained model

We will use **opus-mt-en-swc**, a translation model trained by University of Helsinki. This pretrained model we use, has been pretrained on a larger corpus of English and Kiswahili sentences.



```
]: from transformers import pipeline
   # swahili pretraiined model from hugging face
   model_checkpoint = "Helsinki-NLP/opus-mt-en-swc"
   translator = pipeline("translation", model=model_checkpoint)
```

Let us the test the translation model with some of the English words to see how it is performing before fine tuning.

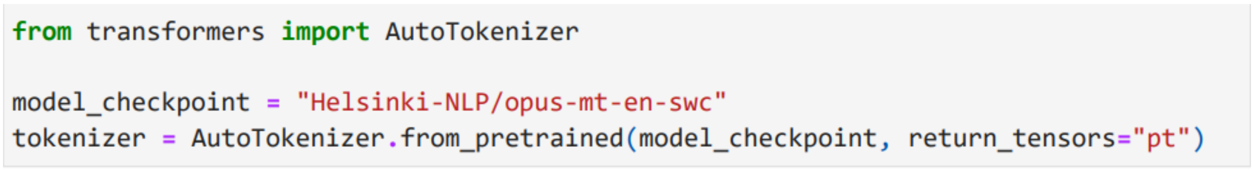


```
: # test pretrained model
  translator("i will not go to school today")

: [{'translation_text': 'Halitaenda shuleni leo'}]
```

The model was able to make predictions but it is not doing well, if you are a Swahili speaker you will notice the grammar is not correct but the meaning of the sentence was unchanged. We’ll perform a more

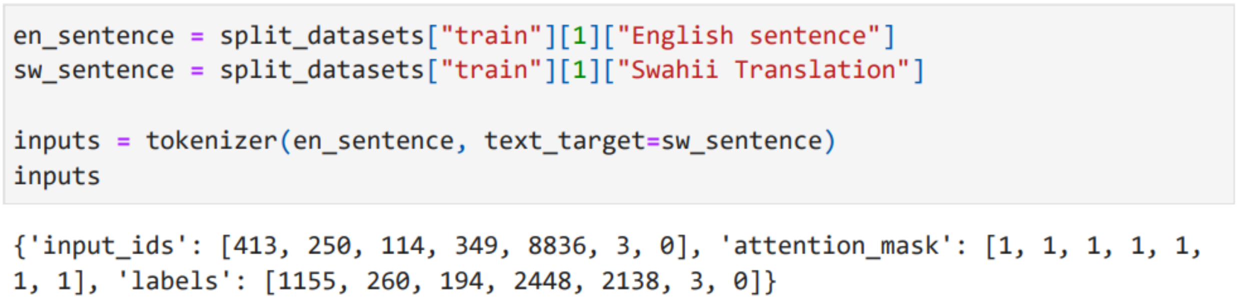
Tokenization of the inputs and targets

The texts all need to be converted into sets of token IDs so the model can make sense of them. For this task, we'll need to tokenize both the inputs and the targets. Our first task is to create our tokenizer object. The same model we are using fine tuning is also used for tokenization.



```python
from transformers import AutoTokenizer

model_checkpoint = "Helsinki-NLP/opus-mt-en-swc"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, return_tensors="pt")
```

A look on the data after tokenization



```python
en_sentence = split_datasets["train"][1]["English sentence"]
sw_sentence = split_datasets["train"][1]["Swahii Translation"]

inputs = tokenizer(en_sentence, text_target=sw_sentence)
inputs
```

```
{'input_ids': [413, 250, 114, 349, 8836, 3, 0], 'attention_mask': [1, 1, 1, 1, 1,
1, 1], 'labels': [1155, 260, 194, 2448, 2138, 3, 0]}
```

Fine Tuning with Keras

First things first, we need an actual model to fine-tune. We'll use the usual AutoModel API:

Data Collation

We'll need a data collator to deal with the padding for dynamic batching. We can't just use a DataCollatorWithPadding because that only pads the inputs (input IDs, attention mask, and token type IDs). Our labels should also be padded to the maximum length encountered in the labels.

Import DataCollatorForSeq2Seq

```
from transformers import DataCollatorForSeq2Seq

data_collator = DataCollatorForSeq2Seq(tokenizer, model=model, return_tensors="tf")
```

We can now use this **data_collator** to convert each of our datasets to a **tf.data.Dataset**, ready for training.

```
tf_train_dataset = model.prepare_tf_dataset(
    tokenized_datasets["train"],
```

```
    collate_fn=data_collator,
    shuffle=True,
    batch_size=32,
)
tf_eval_dataset = model.prepare_tf_dataset(
    tokenized_datasets["test"],
    collate_fn=data_collator,
    shuffle=False,
    batch_size=16,
)
```

Metrics for performance measure

The traditional metric used for translation is the BLEU score. The BLEU score evaluates how close the translations are to their labels. It does not measure the intelligibility or grammatical correctness of the model's generated outputs, but uses statistical rules to ensure that all the words in the generated outputs also appear in the targets. The most commonly used metric for benchmarking translation models today is SacreBLEU, which addresses this weakness (and others) by standardizing the tokenization step.

To use this metric, we first need to install the SacreBLEU library:

```
import evaluate

metric = evaluate.load("sacrebleu")
```

This metrics takes a set of inputs of the translated word and the correct human translated sentence and measure their difference.

Here is an example on Swahili sentences

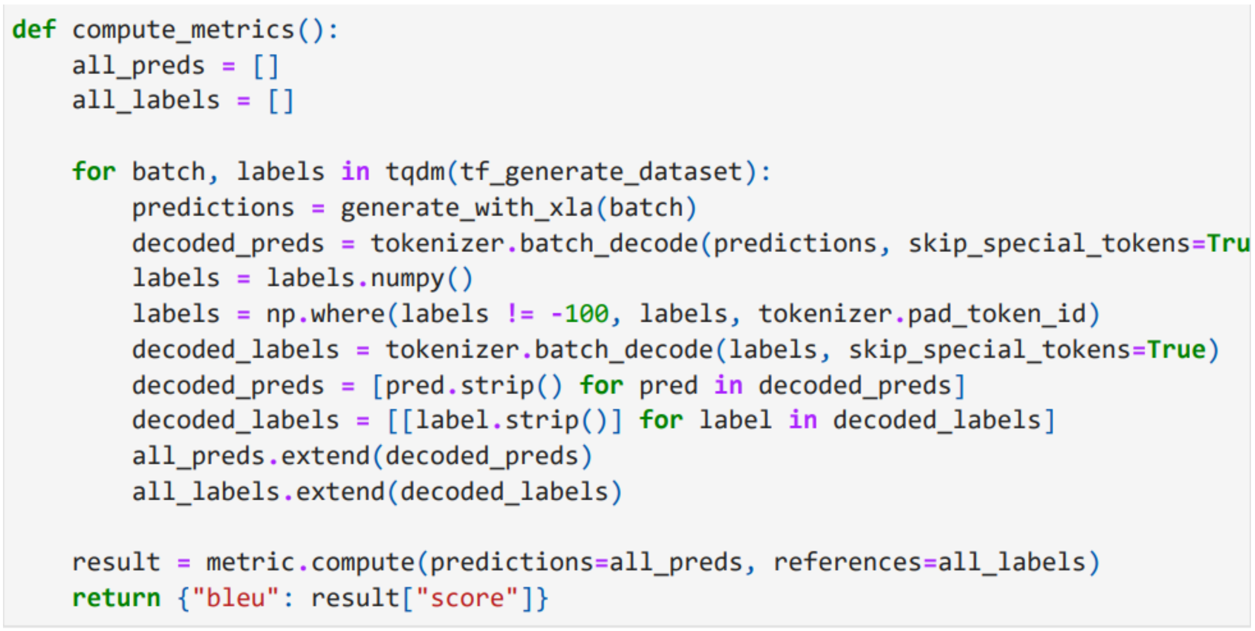
```
# using sacrabeu to
predictions = [
    "Tutaunda ukuta mpya kuzunguka shule yote."
]
references = [
    [
        "Tutatengeneza ukuta mpya kuzunguka shule."
    ]
]
metric.compute(predictions=predictions, references=references)
```

The output of running the above is

```
{'score': 43.47208719449914,
 'counts': [5, 3, 2, 1],
 'totals': [7, 6, 5, 4],
 'precisions': [71.42857142857143, 50.0, 40.0, 25.0],
 'bp': 1.0,
 'sys_len': 7,
 'ref_len': 6}
```

It gets a BLEU score of 43.47 which is rather good compared to other translation models. The score can go from 0 to 100, and higher is better. According to an article written by GOOGLE, Google translate into English had a BLEU score of 55 in 2020. Read more on http://ai.googleblog.com/2020/06/recent-advances-in-google-translate.html/

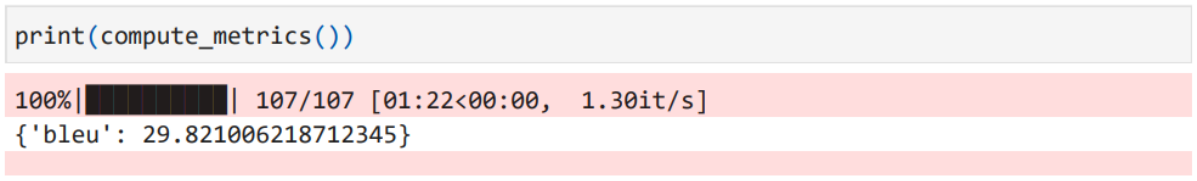
Let’s now create a function to loop through the model and return the score average so that we don’t have to test the model accuracy line by line.



```
def compute_metrics():
    all_preds = []
    all_labels = []

    for batch, labels in tqdm(tf_generate_dataset):
        predictions = generate_with_xla(batch)
        decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=Tru
        labels = labels.numpy()
        labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
        decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
        decoded_preds = [pred.strip() for pred in decoded_preds]
        decoded_labels = [[label.strip()] for label in decoded_labels]
        all_preds.extend(decoded_preds)
        all_labels.extend(decoded_labels)

    result = metric.compute(predictions=all_preds, references=all_labels)
    return {"bleu": result["score"]}
```

Evaluate the accuracy of the model before fine tuning

Before we start, let’s see what kind of results we get from our model without any training



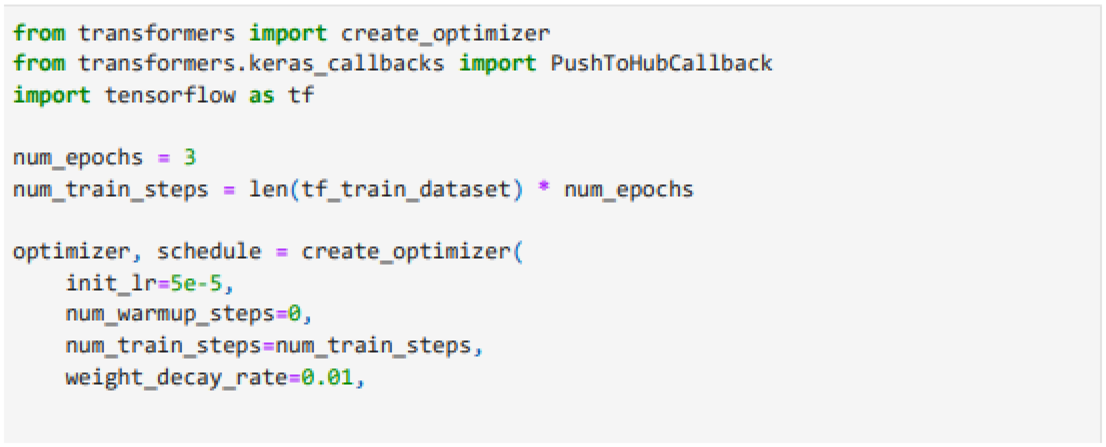
```
print(compute_metrics())
```

```
100%|██████████| 107/107 [01:22<00:00,  1.30it/s]
{'bleu': 29.821006218712345}
```

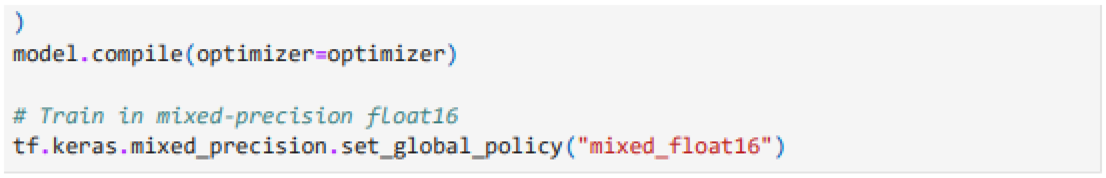
The model achieved an accuracy score of 29.82 which is a little low compared to other translation models. We’ll try to increase this score by atleast 5 points by pre pretraining the model.

Compile Model for Training

Once this is done, we can prepare everything we need to compile and train our model. We use a number of 3 epochs but you can use more to increase accuracy but not too high because it may result to model overfitting. We recommend 10 epochs.



```
from transformers import create_optimizer
from transformers.keras_callbacks import PushToHubCallback
import tensorflow as tf

num_epochs = 3
num_train_steps = len(tf_train_dataset) * num_epochs

optimizer, schedule = create_optimizer(
    init_lr=5e-5,
    num_warmup_steps=0,
    num_train_steps=num_train_steps,
    weight_decay_rate=0.01,
```

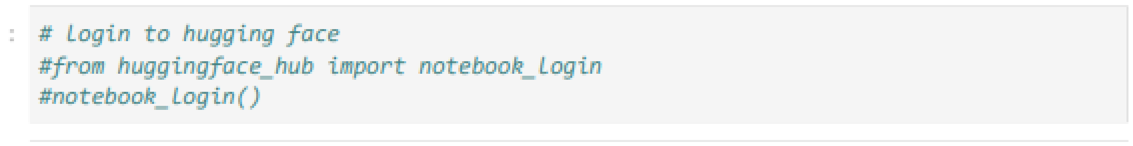


```
)
model.compile(optimizer=optimizer)

# Train in mixed-precision float16
tf.keras.mixed_precision.set_global_policy("mixed_float16")
```

Note the use of **tf.keras.mixed_precision.set_global_policy("mixed_float16")** — this will tell Keras to train using float16, which can give a significant speedup on GPUs that support it (Nvidia 20xx/V100 or newer).

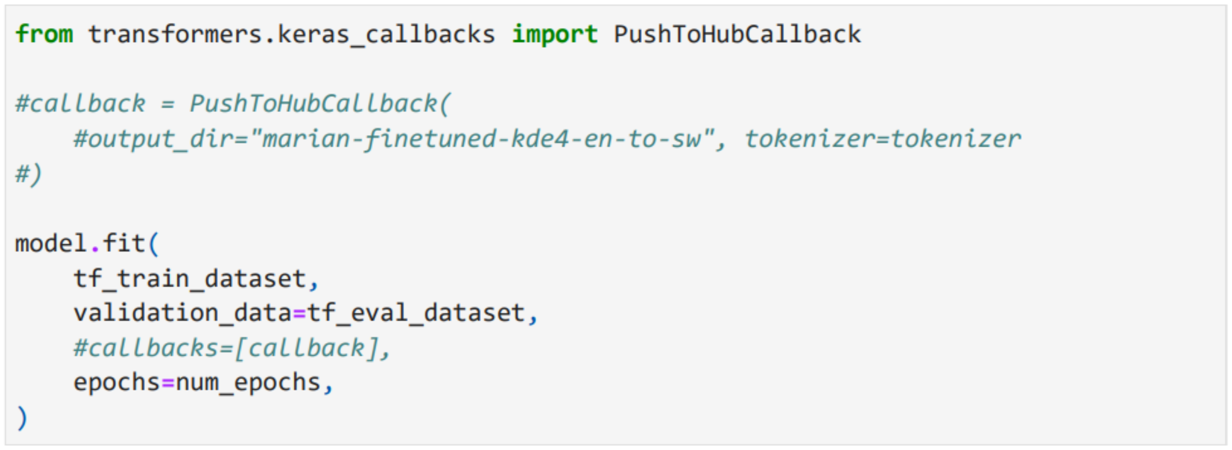
Model training

This is the part where we finally train the model, actually pretraining the model. To save you models so that you don't have to train again, you can create an account on huggingface and uncomment the codes below. To create an account, use this link http://huggingface.co/welcome/



```
# login to hugging face
#from huggingface_hub import notebook_login
#notebook_login()
```

Uncomment the code to login to huggingface but if you don't have an account, you can skip this step.

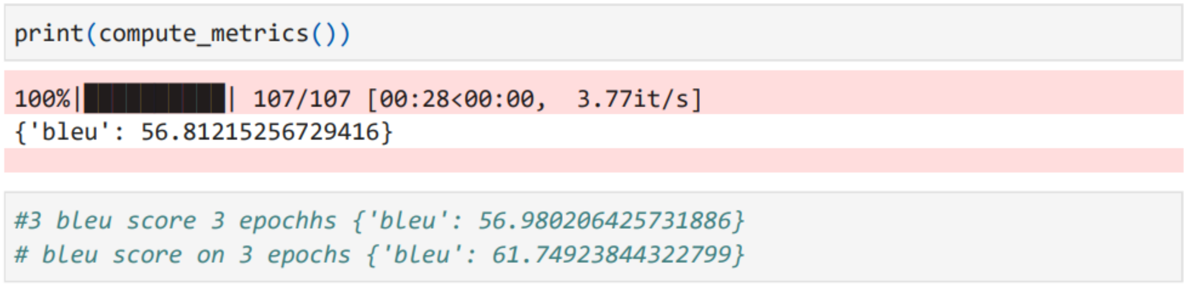


```
from transformers.keras_callbacks import PushToHubCallback

#callback = PushToHubCallback(
    #output_dir="marian-finetuned-kde4-en-to-sw", tokenizer=tokenizer
#)

model.fit(
    tf_train_dataset,
    validation_data=tf_eval_dataset,
    #callbacks=[callback],
    epochs=num_epochs,
)
```

This code will take some time to finish executing but make sure you switched to GPU as your running device. The instruction on how to do it was provided in the other document.

Evaluate the accuracy of the model after fine tuning

Finally, let’s see what our metrics look like now that training has finished.



```
print(compute_metrics())
```

```
100%|██████████| 107/107 [00:28<00:00,  3.77it/s]
{'bleu': 56.81215256729416}
```

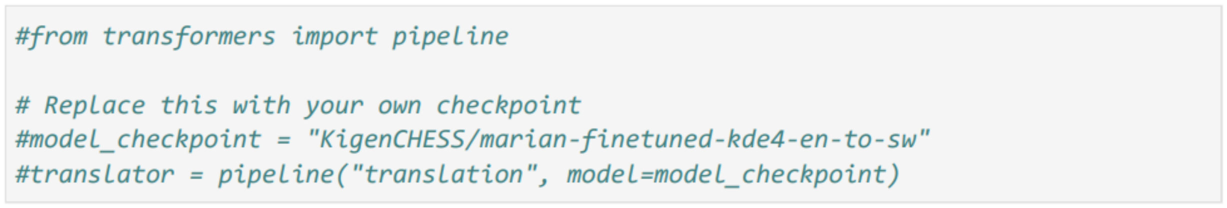
```
#3 bleu score 3 epochhs {'bleu': 56.980206425731886}
# bleu score on 3 epochs {'bleu': 61.74923844322799}
```

You can see the accuracy is now 56 which means the model as done exceptionably well form 29 before fine tuning the model.

Using the pretrained model

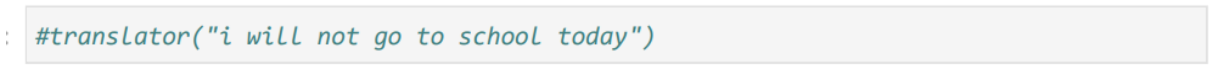
Now that you saved your model, you can load ad use it using a few lines of code. You can also share it with your friends.

If you did not create the account, you can skip this step to deployment



```
#from transformers import pipeline

# Replace this with your own checkpoint
#model_checkpoint = "KigenCHESS/marian-finetuned-kde4-en-to-sw"
#translator = pipeline("translation", model=model_checkpoint)
```

Translation on pretrained model



```
#translator("i will not go to school today")
```

DEPLOYMENT OF THE TRANSLATION MODEL