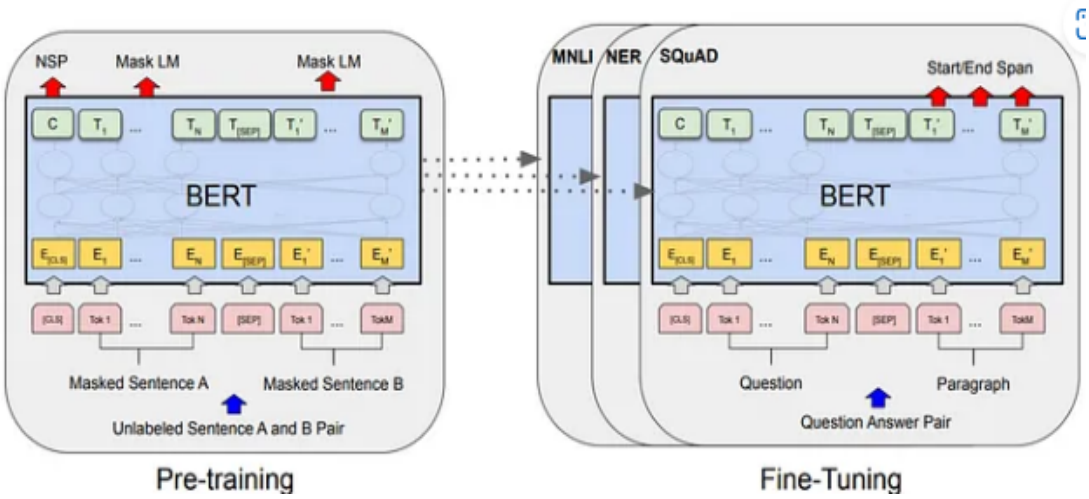
Experiment No 8

Aim: Fine tuning BERT model to perform Natural Language Processing task.

Theory:

BERT

BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers and is a language representation model by Google. It uses two steps, pre-training and fine-tuning, to create state-of-the-art models for a wide range of tasks. Its distinctive feature is the unified architecture across different downstream tasks — what these are, we will discuss soon. That means that the same pre-trained model can be fine-tuned for a variety of final tasks that might not be similar to the task model was trained on and give close to state-of-the-art results.



BERT Architecture

BERT has to differ Architecture BERT Base and BERT Large
BERT Base: L=12, H=768, A=12.
Total Parameters=110M!
BERT Large: L=24, H=1024, A=16.
Total Parameters=340M!!
L = Number of layers (i.e., #Transformer encoder blocks in the stack).
H = Hidden size (i.e. the size of *q, k* and *v* vectors).
A = Number of attention heads.

Pre-training BERT

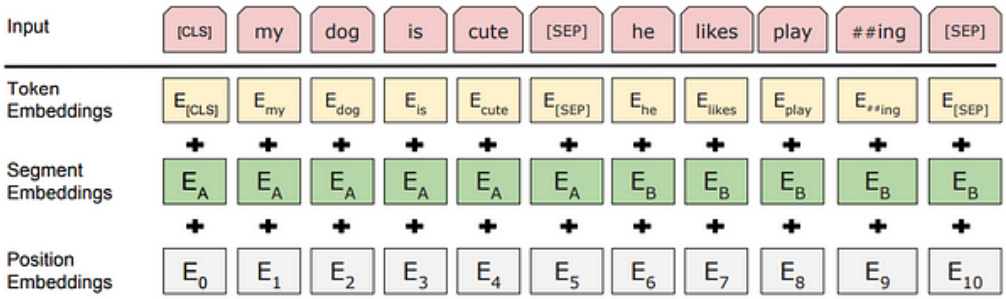
The BERT model is trained on the following two unsupervised tasks.
1. Masked Language Model (MLM)
This task enables the deep bidirectional learning aspect of the model. In this task, some percentage of the input tokens are masked (Replaced with [*MASK*] token) at random and the model tries to predict these masked tokens — not the entire input sequence. The predicted

tokens from the model are then fed into an output softmax over the vocabulary to get the final output words.
This, however creates a mismatch between the pre-training and fine-tuning tasks because the latter does not involve predicting masked words in most of the downstream tasks. This is mitigated by a subtle twist in how we mask the input tokens.
Approximately 15% of the words are masked while training, but all of the masked words are not replaced by the [*MASK*] token.
80% of the time with [*MASK*] tokens.
10% of the time with a random tokens.
10% of the time with the unchanged input tokens that were being masked.

2. Next Sentence Prediction (NSP)

The LM doesn't directly capture the relationship between two sentences which is relevant in many downstream tasks such as Question Answering (QA) and Natural Language Inference (NLI). The model is taught sentence relationships by training on binarized NSP task.
In this task, two sentences — A and B — are chosen for pre-training.
50% of the time B is the actual next sentence that follows A.
50% of the time B is a random sentence from the corpus.
Training — Inputs and Outputs.

The model is trained on both above mentioned tasks simultaneously. This is made possible by clever usage of inputs and outputs.
Inputs



The input representation for BERT

The model needs to take input for both a single sentence or two sentences packed together unambiguously in one token sequence. Authors note that a "sentence" can be an arbitrary span of contiguous text, rather than an actual linguistic sentence. A [SEP] token is used to separate two sentences as well as a using a learnt segment embedding indicating a token as a part of segment A or B.

Problem #1*:* All the inputs are fed in one step — as opposed to RNNs in which inputs are fed sequentially, the model is ***not able to preserve the ordering*** of the input tokens. The order of words in every language is significant, both semantically and syntactically.

Problem #2*:* In order to perform Next Sentence Prediction task properly we need to be able to ***distinguish between sentences A and B***. Fixing the lengths of sentences can be too restrictive and a potential bottleneck for various downstream tasks.

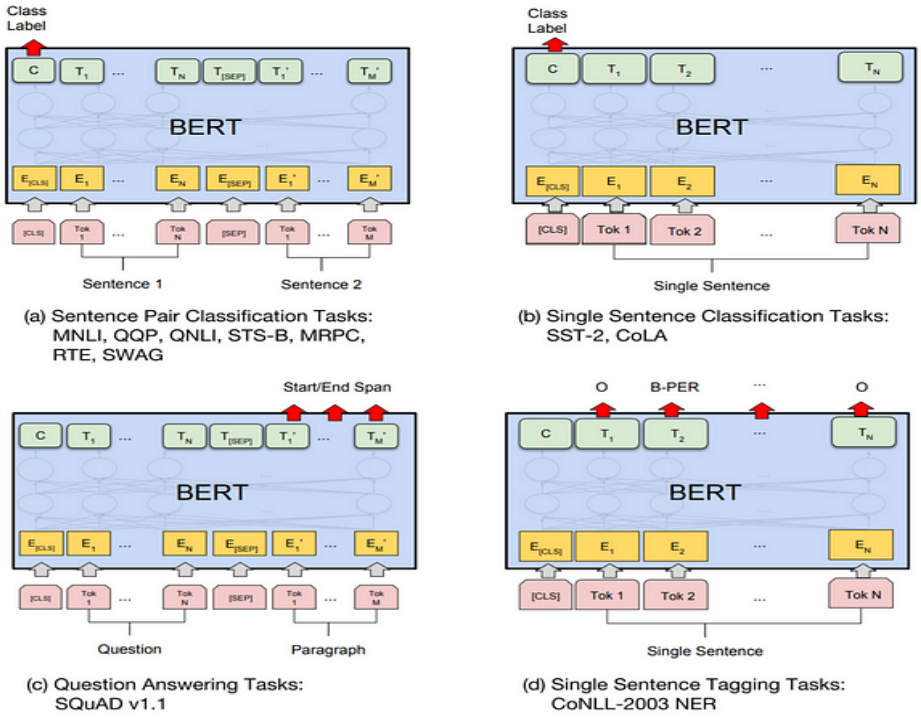
Both of these problems are solved by adding embeddings containing the required information to our original tokens and using the result as the input to our BERT model. The following embeddings are added to token embeddings:

Segment Embedding: They provide information about the sentence a particular token is a part of.

Position Embedding: They provide information about the order of words in the input.
Outputs

Fine-tuning BERT

Fine-tuning on various downstream tasks is done by swapping out the appropriate inputs or outputs. In the general run of things, to train task-specific models, we add an extra output layer to existing BERT and fine-tune the resultant model — all parameters, end to end. A positive consequence of adding layers — input/output and not changing the BERT model is that only a minimal number of parameters need to be learned from scratch making the procedure fast, cost and resource efficient.
Just to give you an idea of how fast and efficient it is, the authors claim that all the results in the paper can be replicated in *at most 1 hour* on a *single Cloud TPU*, or *a few hours on a GPU*, starting from the exact same pre-trained model.



Fine-tuning BERT on various downstream tasks.

In Sentence Pair Classification and Single Sentence Classification, the final state corresponding to [*CLS*] token is used as input for the additional layers that makes the prediction.
In QA tasks, a start (S) and an end (E) vector are introduced during fine tuning. The question is fed as sentence A and the answer as sentence B. The probability of word *i* being the start of the answer span is computed as a dot product between T*i* (final state corresponding to *i*th input

token) and S (start vector) followed by a softmax over all of the words in the paragraph. A similar method is used for end span.

Advantages of Fine-Tuning

Quicker Development

First, the pre-trained BERT model weights already encode a lot of information about our language. As a result, it takes much less time to train our fine-tuned model - it is as if we have already trained the bottom layers of our network extensively and only need to gently tune them while using their output as features for our classification task. In fact, the authors recommend only 2-4 epochs of training for fine-tuning BERT on a specific NLP task (compared to the hundreds of GPU hours needed to train the original BERT model or a LSTM from scratch!).

Less Data

In addition, and perhaps just as important, because of the pre-trained weights this method allows us to fine-tune our task on a much smaller dataset than would be required in a model that is built from scratch. A major drawback of NLP models built from scratch is that we often need a prohibitively large dataset in order to train our network to reasonable accuracy, meaning a lot of time and energy had to be put into dataset creation. By fine-tuning BERT, we are now able to get away with training a model to good performance on a much smaller amount of training data.

Better Results

Finally, this simple fine-tuning procedure (typically adding one fully-connected layer on top of BERT and training for a few epochs) was shown to achieve state of the art results with minimal task-specific adjustments for a wide variety of tasks: classification, language inference, semantic similarity, question answering, etc. Rather than implementing custom and sometimes-obscure architectures shown to work well on a specific task, simply fine-tuning BERT is shown to be a better (or at least equal) alternative.

Steps to Fine Tune BERT Model to perform Multi Class Text Classification

1. **Load dataset**
2. **Pre-process data**
3. **Define model**
4. **Train the model**
5. **Evaluate**

Lab Exercise to be Performed in this Session:

Perform Multi Label Text Classification by Fine tuning BERT model.

Fine-tuning BERT (and friends) for multi-label text classification

In this notebook, we are going to fine-tune BERT to predict one or more labels for a given piece of text. Note that this notebook illustrates how to fine-tune a bert-base-uncased model, but you can also fine-tune a RoBERTa, DeBERTa, DistilBERT, CANINE, ... checkpoint in the same way.

All of those work in the same way: they add a linear layer on top of the base model, which is used to produce a tensor of shape (batch_size, num_labels), indicating the unnormalized scores for a number of labels for every example in the batch.

Set-up environment

First, we install the libraries which we'll use: HuggingFace Transformers and Datasets.

```
!pip install -q transformers datasets
```

Load dataset

Next, let's download a multi-label text classification dataset from the hub.

At the time of writing, I picked a random one as follows:

- first, go to the "datasets" tab on huggingface.co
- next, select the "multi-label-classification" tag on the left as well as the the "1k<10k" tag (fo find a relatively small dataset).

Note that you can also easily load your local data (i.e. csv files, txt files, Parquet files, JSON, ...) as explained here.

```
from datasets import load_dataset

dataset = load_dataset("sem_eval_2018_task_1", "subtask5.english")
```

```
Reusing dataset sem_eval2018_task1 (/root/.cache/huggingface/datasets/sem_eval2018_task1/subtask5.english/1.1.0/c8af6e4a
100%                                                3/3 [00:00<00:00, 75.93it/s]
```

As we can see, the dataset contains 3 splits: one for training, one for validation and one for testing.

```
dataset
```

```
DatasetDict({
    train: Dataset({
        features: ['ID', 'Tweet', 'anger', 'anticipation', 'disgust', 'fear', 'joy', 'love', 'optimism', 'pessimism',
'sadness', 'surprise', 'trust'],
        num_rows: 6838
    })
    test: Dataset({
        features: ['ID', 'Tweet', 'anger', 'anticipation', 'disgust', 'fear', 'joy', 'love', 'optimism', 'pessimism',
'sadness', 'surprise', 'trust'],
        num_rows: 3259
    })
    validation: Dataset({
        features: ['ID', 'Tweet', 'anger', 'anticipation', 'disgust', 'fear', 'joy', 'love', 'optimism', 'pessimism',
'sadness', 'surprise', 'trust'],
        num_rows: 886
    })
})
```

Let's check the first example of the training split:

```
example = dataset['train'][0]
example
```

```
{'ID': '2017-En-21441',
 'Tweet': "“Worry is a down payment on a problem you may never have'. \xa0Joyce Meyer.  #motivation #leadership
#worry",
 'anger': False,
 'anticipation': True,
 'disgust': False,
 'fear': False,
 'joy': False,
 'love': False,
 'optimism': True,
 'pessimism': False,
 'sadness': False,
 'surprise': False,
 'trust': True}
```

The dataset consists of tweets, labeled with one or more emotions.

Let's create a list that contains the labels, as well as 2 dictionaries that map labels to integers and back.

```
labels = [label for label in dataset['train'].features.keys() if label not in ['ID', 'Tweet']]
id2label = {idx:label for idx, label in enumerate(labels)}
label2id = {label:idx for idx, label in enumerate(labels)}
labels
```

```
['anger',
 'anticipation',
 'disgust',
 'fear',
 'joy',
 'love',
 'optimism',
 'pessimism',
 'sadness',
 'surprise',
 'trust']
```

Preprocess data

As models like BERT don't expect text as direct input, but rather `input_ids`, etc., we tokenize the text using the tokenizer. Here I'm using the `AutoTokenizer` API, which will automatically load the appropriate tokenizer based on the checkpoint on the hub.

What's a bit tricky is that we also need to provide labels to the model. For multi-label text classification, this is a matrix of shape (batch_size, num_labels). Also important: this should be a tensor of floats rather than integers, otherwise PyTorch' `BCEWithLogitsLoss` (which the model will use) will complain, as explained here.

```
from transformers import AutoTokenizer
import numpy as np

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

def preprocess_data(examples):
  # take a batch of texts
  text = examples["Tweet"]
  # encode them
  encoding = tokenizer(text, padding="max_length", truncation=True, max_length=128)
  # add labels
  labels_batch = {k: examples[k] for k in examples.keys() if k in labels}
  # create numpy array of shape (batch_size, num_labels)
  labels_matrix = np.zeros((len(text), len(labels)))
  # fill numpy array
  for idx, label in enumerate(labels):
    labels_matrix[:, idx] = labels_batch[label]

  encoding["labels"] = labels_matrix.tolist()

  return encoding


encoded_dataset = dataset.map(preprocess_data, batched=True, remove_columns=dataset['train'].column_names)
```

```
Loading cached processed dataset at /root/.cache/huggingface/datasets/sem_eval2018_task1/subtask5.english/1.1.0/c8af6e4a
Loading cached processed dataset at /root/.cache/huggingface/datasets/sem_eval2018_task1/subtask5.english/1.1.0/c8af6e4a
Loading cached processed dataset at /root/.cache/huggingface/datasets/sem_eval2018_task1/subtask5.english/1.1.0/c8af6e4a
```

```
example = encoded_dataset['train'][0]
print(example.keys())
```

```
dict_keys(['attention_mask', 'input_ids', 'labels', 'token_type_ids'])
```

```
tokenizer.decode(example['input_ids'])
```

```
'[CLS] “ worry is a down payment on a problem you may never have '. joyce mey
er. # motivation # leadership # worry [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PA
D] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [P
AD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
```

```
example['labels']
```

```
tensor([0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 1.])
```

```
[id2label[idx] for idx, label in enumerate(example['labels']) if label == 1.0]
```

```
['anticipation', 'optimism', 'trust']
```

Finally, we set the format of our data to PyTorch tensors. This will turn the training, validation and test sets into standard PyTorch datasets.

```
encoded_dataset.set_format("torch")
```

Define model

Here we define a model that includes a pre-trained base (i.e. the weights from bert-base-uncased) are loaded, with a random initialized classification head (linear layer) on top. One should fine-tune this head, together with the pre-trained base on a labeled dataset.

This is also printed by the warning.

We set the `problem_type` to be "multi_label_classification", as this will make sure the appropriate loss function is used (namely `BCEWithLogitsLoss`). We also make sure the output layer has `len(labels)` output neurons, and we set the id2label and label2id mappings.

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased",
                                                           problem_type="multi_label_classification",
                                                           num_labels=len(labels),
                                                           id2label=id2label,
                                                           label2id=label2id)
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertForSequenceClassification:
- This IS expected if you are initializing BertForSequenceClassification from the checkpoint of a model trained on anoth
- This IS NOT expected if you are initializing BertForSequenceClassification from the checkpoint of a model that you exp
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and ar
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

Train the model!

We are going to train the model using HuggingFace's Trainer API. This requires us to define 2 things:

- `TrainingArguments`, which specify training hyperparameters. All options can be found in the docs. Below, we for example specify that we want to evaluate after every epoch of training, we would like to save the model every epoch, we set the learning rate, the batch size to use for training/evaluation, how many epochs to train for, and so on.
- a `Trainer` object (docs can be found here).

```
batch_size = 8
metric_name = "f1"
```

```
from transformers import TrainingArguments, Trainer

args = TrainingArguments(
    f"bert-finetuned-sem_eval-english",
    evaluation_strategy = "epoch",
    save_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=5,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model=metric_name,
    #push_to_hub=True,
)
```

We are also going to compute metrics while training. For this, we need to define a `compute_metrics` function, that returns a dictionary with the desired metric values.

```
from sklearn.metrics import f1_score, roc_auc_score, accuracy_score
from transformers import EvalPrediction
import torch

# source: https://jesusleal.io/2021/04/21/Longformer-multilabel-classification/
def multi_label_metrics(predictions, labels, threshold=0.5):
    # first, apply sigmoid on predictions which are of shape (batch_size, num_labels)
    sigmoid = torch.nn.Sigmoid()
    probs = sigmoid(torch.Tensor(predictions))
    # next, use threshold to turn them into integer predictions
    y_pred = np.zeros(probs.shape)
    y_pred[np.where(probs >= threshold)] = 1
    # finally, compute metrics
    y_true = labels
    f1_micro_average = f1_score(y_true=y_true, y_pred=y_pred, average='micro')
    roc_auc = roc_auc_score(y_true, y_pred, average = 'micro')
    accuracy = accuracy_score(y_true, y_pred)
    # return as dictionary
    metrics = {'f1': f1_micro_average,
               'roc_auc': roc_auc,
               'accuracy': accuracy}
    return metrics

def compute_metrics(p: EvalPrediction):
    preds = p.predictions[0] if isinstance(p.predictions,
            tuple) else p.predictions
    result = multi_label_metrics(
        predictions=preds,
        labels=p.label_ids)
    return result
```

Let's verify a batch as well as a forward pass:

```
encoded_dataset['train'][0]['labels'].type()
```

```
'torch.FloatTensor'
```

```
encoded_dataset['train']['input_ids'][0]
```

```
tensor([  101,  1523,  4737,  2003,  1037,  2091,  7909,  2006,  1037,  3291,
         2017,  2089,  2196,  2031,  1005,  1012, 11830, 11527,  1012,  1001,
        14354,  1001,  4105,  1001,  4737,   102,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0])
```

```
#forward pass
outputs = model(input_ids=encoded_dataset['train']['input_ids'][0].unsqueeze(0), labels=encoded_dataset['train'][0]['labels'
outputs
```

```
SequenceClassifierOutput([('loss',
                           tensor(0.6442, grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)),
                          ('logits',
                           tensor([[-0.2136,  0.1645, -0.2064, -0.0940, -0.5121,  0.0718,  0.1167, -0.1767,
                                    0.0968,  0.0115, -0.1126]], grad_fn=<AddmmBackward0>))])
```

Let's start training!

```
trainer = Trainer(
    model,
    args,
    train_dataset=encoded_dataset["train"],
    eval_dataset=encoded_dataset["validation"],
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)
```

```
trainer.train()
```

```
***** Running training *****
  Num examples = 6838
  Num Epochs = 5
  Instantaneous batch size per device = 8
  Total train batch size (w. parallel, distributed & accumulation) = 8
  Gradient Accumulation steps = 1
  Total optimization steps = 4275
```

[4275/4275 11:16, Epoch 5/5]

Epoch	Training Loss	Validation Loss	F1	Roc Auc	Accuracy
1	0.279300	0.315630	0.690058	0.786618	0.262980
2	0.237500	0.308757	0.702635	0.795974	0.273138
3	0.202900	0.319157	0.709615	0.805806	0.279910
4	0.180800	0.324087	0.702689	0.798627	0.277652
5	0.157700	0.329103	0.701452	0.798265	0.272009

```
***** Running Evaluation *****
  Num examples = 886
  Batch size = 8
Saving model checkpoint to bert-finetuned-sem_eval-english/checkpoint-855
Configuration saved in bert-finetuned-sem_eval-english/checkpoint-855/config.json
Model weights saved in bert-finetuned-sem_eval-english/checkpoint-855/pytorch_model.bin
tokenizer config file saved in bert-finetuned-sem_eval-english/checkpoint-855/tokenizer_config.json
Special tokens file saved in bert-finetuned-sem_eval-english/checkpoint-855/special_tokens_map.json
***** Running Evaluation *****
  Num examples = 886
  Batch size = 8
Saving model checkpoint to bert-finetuned-sem_eval-english/checkpoint-1710
Configuration saved in bert-finetuned-sem_eval-english/checkpoint-1710/config.json
Model weights saved in bert-finetuned-sem_eval-english/checkpoint-1710/pytorch_model.bin
tokenizer config file saved in bert-finetuned-sem_eval-english/checkpoint-1710/tokenizer_config.json
Special tokens file saved in bert-finetuned-sem_eval-english/checkpoint-1710/special_tokens_map.json
***** Running Evaluation *****
  Num examples = 886
  Batch size = 8
Saving model checkpoint to bert-finetuned-sem_eval-english/checkpoint-2565
Configuration saved in bert-finetuned-sem_eval-english/checkpoint-2565/config.json
Model weights saved in bert-finetuned-sem_eval-english/checkpoint-2565/pytorch_model.bin
tokenizer config file saved in bert-finetuned-sem_eval-english/checkpoint-2565/tokenizer_config.json
Special tokens file saved in bert-finetuned-sem_eval-english/checkpoint-2565/special_tokens_map.json
***** Running Evaluation *****
  Num examples = 886
  Batch size = 8
Saving model checkpoint to bert-finetuned-sem_eval-english/checkpoint-3420
Configuration saved in bert-finetuned-sem_eval-english/checkpoint-3420/config.json
Model weights saved in bert-finetuned-sem_eval-english/checkpoint-3420/pytorch_model.bin
tokenizer config file saved in bert-finetuned-sem_eval-english/checkpoint-3420/tokenizer_config.json
Special tokens file saved in bert-finetuned-sem_eval-english/checkpoint-3420/special_tokens_map.json
***** Running Evaluation *****
  Num examples = 886
  Batch size = 8
```

Evaluate

After training, we evaluate our model on the validation set.

```
trainer.evaluate()
```

```
***** Running Evaluation *****
  Num examples = 886
  Batch size = 8
```

[408/408 02:01]

```
{'epoch': 5.0,
 'eval_accuracy': 0.2799097065462754,
 'eval_f1': 0.7096149188665537,
 'eval_loss': 0.31915703415870667,
 'eval_roc_auc': 0.805805895058838,
 'eval_runtime': 4.7187,
 'eval_samples_per_second': 187.766,
 'eval_steps_per_second': 23.524}
```

Inference

Let's test the model on a new sentence:

```
text = "I'm happy I can finally train a model for multi-label classification"

encoding = tokenizer(text, return_tensors="pt")
encoding = {k: v.to(trainer.model.device) for k,v in encoding.items()}

outputs = trainer.model(**encoding)
```

The logits that come out of the model are of shape (batch_size, num_labels). As we are only forwarding a single sentence through the model, the `batch_size` equals 1. The logits is a tensor that contains the (unnormalized) scores for every individual label.

```
logits = outputs.logits
logits.shape
```

```
torch.Size([1, 11])
```

To turn them into actual predicted labels, we first apply a sigmoid function independently to every score, such that every score is turned into a number between 0 and 1, that can be interpreted as a "probability" for how certain the model is that a given class belongs to the input text.

Next, we use a threshold (typically, 0.5) to turn every probability into either a 1 (which means, we predict the label for the given example) or a 0 (which means, we don't predict the label for the given example).

```
# apply sigmoid + threshold
sigmoid = torch.nn.Sigmoid()
probs = sigmoid(logits.squeeze().cpu())
predictions = np.zeros(probs.shape)
predictions[np.where(probs >= 0.5)] = 1
# turn predicted id's into actual label names
predicted_labels = [id2label[idx] for idx, label in enumerate(predictions) if label == 1.0]
print(predicted_labels)
```

```
['joy', 'optimism']
```