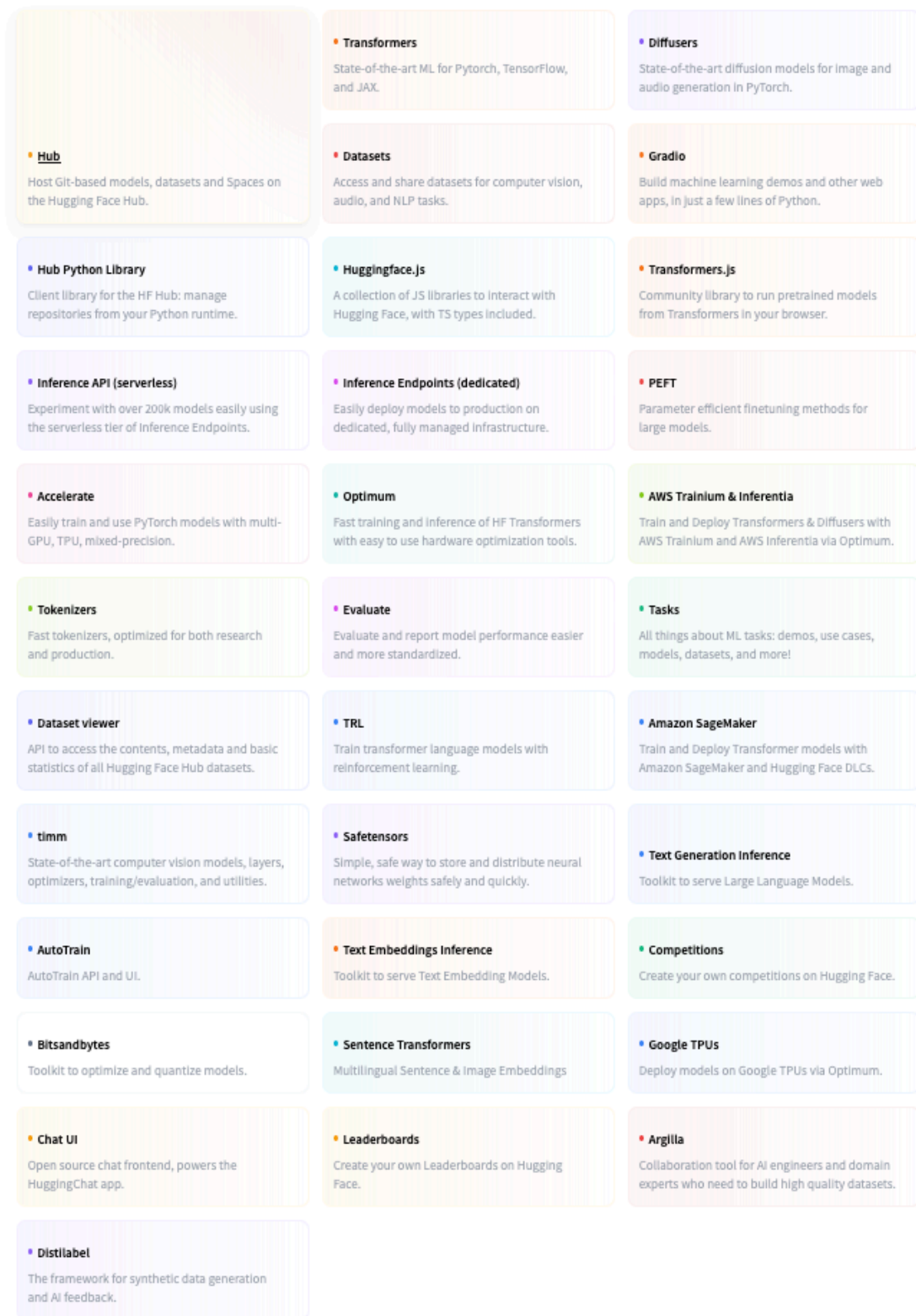


# Introduction to Huggingface 🤗

Huggingface is an open source data science and machine learning platform for AI builders. It has a lot of useful libraries which can abstract a significant amount of source code and allow efficient usage of artifacts like datasets, models, evaluation metrics, etc.

In addition to libraries, it also provides Hub and Spaces which are collaboration platform and an interactive GUI respectively.



We will be focusing only on some key functionalities of useful libraries of Huggingface with Pytorch implementation of artifacts, which are outlined as follows -

1. Datasets
2. Transformers
3. Evaluate

## 4. Examples

Some other key libraries for NLP tasks that you can explore - Tokenizers, Sentence Transformers, PEFT, Diffusers, Bitsandbytes, Safetensors, Accelerate

Assignments will mostly not involve Huggingface. This will be more helpful for projects where you can focus on research ideas without getting into the abstractions of base implementation of models.

```
In [1]: cache_dir = './huggingface_cache'
```

### Disclaimer:

This tutorial is designed to introduce you to HuggingFace's tools, including datasets, transformers, and the evaluate library, in NLP workflows. The focus here is on practical implementation rather than in-depth explanations of the underlying concepts. While concepts are not covered in this tutorial, rest assured that they will be thoroughly explained in lectures. Even if you don't fully understand everything at this stage, this tutorial will still be valuable for your exposure, your projects, and will serve as a helpful reference when revisiting how to implement these concepts after they've been covered in class. Through the course, you'll gain a full conceptual grasp of the models and techniques used, and you can come back to this notebook to see just how to implement them using HuggingFace.

## 1. Datasets

The Huggingface Datasets library is an open-source Python library designed to simplify the process of accessing, managing, and processing a wide range of datasets. The library is optimized for performance, allowing users to load, filter, and transform datasets efficiently, even when working with large-scale data.

Datasets can be accessed with just a few lines of code, and the library handles tasks like downloading, caching, and versioning automatically. The library includes powerful tools for data processing, allowing users to easily manipulate and transform datasets. This includes tasks like filtering, shuffling, splitting, and mapping functions over dataset elements.

Docs: <https://huggingface.co/docs/datasets>

```
In [2]: # !pip install datasets
```

```
In [3]: from datasets import load_dataset
```

```
In [4]: dataset = load_dataset('ccdv/patent-classification', cache_dir=cache_dir)
```

```
README.md: 0.00B [00:00, ?B/s]
train-00000-of-00001.parquet: 0%|          | 0.00/194M [00:00<?, ?B/s]
validation-00000-of-00001.parquet: 0%|          | 0.00/39.5M [00:00<?, ?B/s]
test-00000-of-00001.parquet: 0%|          | 0.00/39.1M [00:00<?, ?B/s]
Generating train split: 0%|          | 0/25000 [00:00<?, ? examples/s]
Generating validation split: 0%|          | 0/5000 [00:00<?, ? examples/s]
Generating test split: 0%|          | 0/5000 [00:00<?, ? examples/s]
```

## Suggestions

If you don't specify a cache directory, all files will be downloaded to your conda environment's root, drastically increasing its size. Specifying a `cache_dir` is very helpful, particularly in distributed environments, as you can control the location where these large datasets (and models) are stored, without impacting your environment.

```
In [5]: dataset
```

```
Out[5]: DatasetDict({
  train: Dataset({
    features: ['text', 'label'],
    num_rows: 25000
  })
  validation: Dataset({
    features: ['text', 'label'],
    num_rows: 5000
  })
  test: Dataset({
    features: ['text', 'label'],
    num_rows: 5000
  })
})
```

```
In [6]: dataset_train = load_dataset('ccdv/patent-classification', cache_dir=cache_d
```

```
In [7]: dataset_train
```

```
Out[7]: Dataset({
  features: ['text', 'label'],
  num_rows: 25000
})
```

```
In [8]: print('Text -', dataset_train[0]['text'][0:300], '...(continued)' # trimmed
        print('Label -', dataset_train[0]['label'])
```

Text - turning now to the drawings , there is shown in fig1 an integrated circuit continuity testing system in which a specimen or circuit configuration 16 is mounted on a fixture 18 operable to vibrate the specimen under controlled conditions , e . g . sinusoidally , randomly , or a combination of the two ... (continued)

Label - 6

```
In [9]: # Get all the labels directly
        label_info = dataset['train'].features['label']
        print(label_info)
```

ClassLabel(names=['Human Necessities', 'Performing Operations; Transporting', 'Chemistry; Metallurgy', 'Textiles; Paper', 'Fixed Constructions', 'Mechanical Engineering; Lightning; Heating; Weapons; Blasting', 'Physics', 'Electricity', 'General tagging of new or cross-sectional technology'], id=None)

```
In [10]: # Get a mapping of label index to name and vice-versa
        id2label = {i: label for i, label in enumerate(label_info.names)}
        label2id = {label: i for i, label in enumerate(label_info.names)}

        print("id2label:", id2label)
        print("label2id:", label2id)
```

```
id2label: {0: 'Human Necessities', 1: 'Performing Operations; Transporting',
2: 'Chemistry; Metallurgy', 3: 'Textiles; Paper', 4: 'Fixed Constructions',
5: 'Mechanical Engineering; Lightning; Heating; Weapons; Blasting', 6: 'Physics', 7: 'Electricity', 8: 'General tagging of new or cross-sectional technology'}
label2id: {'Human Necessities': 0, 'Performing Operations; Transporting': 1,
'Chemistry; Metallurgy': 2, 'Textiles; Paper': 3, 'Fixed Constructions': 4,
'Mechanical Engineering; Lightning; Heating; Weapons; Blasting': 5, 'Physics': 6, 'Electricity': 7, 'General tagging of new or cross-sectional technology': 8}
```

```
In [11]: from torch.utils.data import DataLoader
```

```
data_loader = DataLoader(dataset_train, batch_size=10, shuffle=True)
# can add a custom sampler, distributed sampler, etc., like with a standard
```

```
In [25]: device = 'cuda'
```

```
for idx, batch in enumerate (data_loader):
    print(f'\nShowing Batch {idx} of {len(data_loader)}')
    print('Data type -', type(batch))
    print('Keys -', batch.keys())
    print('Batch labels -', batch['label']) # only displaying the label for b
    print('Device -', batch['label'].device)
    print('Moving tensor to GPU')
    batch['label'] = batch['label'].to(device=device) # device='cuda' for Nv
    print('Device after changing -', batch['label'].device)
    print('\n-----END OF BATCH-----')
    if idx >= 3:
        break # to break the loop as it is there only for demonstration
```

```

Showing Batch 0 of 2500
Data type - <class 'dict'>
Keys - dict_keys(['text', 'label'])
Batch labels - tensor([7, 6, 6, 0, 2, 1, 2, 5, 1, 6])
Device - cpu
Moving tensor to GPU
Device after changing - cuda:0

```

-----END OF BATCH-----

```

Showing Batch 1 of 2500
Data type - <class 'dict'>
Keys - dict_keys(['text', 'label'])
Batch labels - tensor([6, 7, 6, 7, 5, 0, 5, 6, 1, 5])
Device - cpu
Moving tensor to GPU
Device after changing - cuda:0

```

-----END OF BATCH-----

```

Showing Batch 2 of 2500
Data type - <class 'dict'>
Keys - dict_keys(['text', 'label'])
Batch labels - tensor([7, 8, 8, 7, 0, 1, 4, 0, 7, 5])
Device - cpu
Moving tensor to GPU
Device after changing - cuda:0

```

-----END OF BATCH-----

```

Showing Batch 3 of 2500
Data type - <class 'dict'>
Keys - dict_keys(['text', 'label'])
Batch labels - tensor([6, 1, 0, 0, 6, 7, 2, 8, 4, 1])
Device - cpu
Moving tensor to GPU
Device after changing - cuda:0

```

-----END OF BATCH-----

## Common pitfalls

- Non-numeric data is not automatically converted into tensor by the dataloader. They have to be dealt with separately. As a result, tensor functions also don't work with them.
- `.to(device)` is not an in-place operation. If it is not assigned to an object, the original object doesn't move to GPU, causing a device mismatch error (if your model is on GPU).

## 2. Transformers

Hugging Face Transformers is an open-source Python library that offers a vast collection of **pre-trained Transformer models** for tasks in natural language processing (NLP), computer vision, audio processing, and more. It streamlines the implementation of Transformer models by handling the complexities of training or deploying models, allowing users to work with higher-level APIs instead of directly using frameworks like PyTorch, TensorFlow, or JAX.

Docs: <https://huggingface.co/docs/transformers>

You can utilize them directly for application, in a pipeline, or further training/fine-tuning.

Usually, each model has two essential components - tokenizer and model.

## 2.1. Tokenizers

A tokenizer is in charge of preparing the inputs for a model. The library contains tokenizers for all the models.

Tokenizer API Docs:

[https://huggingface.co/docs/transformers/main\\_classes/tokenizer](https://huggingface.co/docs/transformers/main_classes/tokenizer)

```
In [26]: from transformers import AutoTokenizer
```

```
In [27]: bert_tokenizer = AutoTokenizer.from_pretrained("textattack/bert-base-uncased")
```

```
In [28]: import random

vocab = bert_tokenizer.get_vocab() # a dictionary of token:index pairs
vocab_keys = list(vocab.keys())
vocab_size = len(vocab)
print('Size of vocabulary=', vocab_size)
print('Some example of key-value pairs:')
for idx in range(20): # display some random examples
    random_index = random.randint(0, vocab_size)
    random_token = vocab_keys[random_index]
    print(random_token, ': ', vocab[random_token])
```



```

Size of vocabulary= 30522
Some example of key-value pairs:
sable : 23492
francesco : 11400
rt : 19387
offered : 3253
1265 : 7
starter : 11753
nursery : 13640
##aint : 22325
cincinnati : 7797
911 : 19989
judy : 12120
beautifully : 17950
girls : 3057
dummy : 24369
literacy : 8433
greenville : 20967
lennox : 21060
uncomfortable : 8796
relativity : 20805
tail : 5725

```

```
In [29]: input_text = 'Hello, welcome to CS 7650 - Natural Language. We hope you have
```

```
In [30]: tokens = bert_tokenizer(input_text)
print(tokens['input_ids'])
print('Type-', type(tokens['input_ids']))
```

```

[101, 7592, 1010, 6160, 2000, 20116, 6146, 12376, 1011, 3019, 2653, 1012, 205
7, 3246, 2017, 2031, 4569, 1012, 102]
Type- <class 'list'>

```

```
In [31]: # Let's try to reproduce the string using the tokens.
# Ideally, you should be able to retrieve the same string if the key-value m
bert_tokenizer.decode(tokens['input_ids'])
```

```
Out[31]: '[CLS] hello, welcome to cs 7650 - natural language. we hope you have fun.
[SEP]'
```

## Every Model Has Its Own Tokenizer

Every model in natural language processing (NLP) typically has its own tokenizer because the tokenizer is tightly coupled with the model's architecture, vocabulary, and training process. The design of the tokenizer affects how the model interprets input text and, consequently, its performance on various tasks. Here's a detailed explanation of why this is the case:

### 1. Vocabulary Alignment

- **Vocabulary Size:** Different models have different vocabulary sizes, which is the set of tokens (words, subwords, or characters) the model understands. A tokenizer must be designed to align with this specific vocabulary.

- **Predefined Tokens:** Models are trained with specific tokens representing words, subwords, or special tokens like `[CLS]`, `[SEP]`, and `[PAD]`. The tokenizer needs to ensure that the input text is split into these predefined tokens.

## 2. Handling of Special Tokens

- Models use special tokens to handle different types of inputs, manage sequences, or denote the start and end of sentences. For example:
  - BERT uses `[CLS]` for classification and `[SEP]` to separate sentences.
  - GPT uses `<|endoftext|>` to denote the end of text.
- The tokenizer must insert these tokens correctly in the input text based on how the model was trained.

### Illustration

Consider BERT and GPT-2

If you used BERT's tokenizer with GPT-2 or vice versa, the models would not interpret the inputs correctly, leading to inaccurate or nonsensical outputs.

```
In [32]: from transformers import GPT2Tokenizer

gpt_2_tokenizer = GPT2Tokenizer.from_pretrained("gpt2", cache_dir=cache_dir)
```

```
In [33]: # Decoding back the same sequence of the above string using a GPT-2 tokenizer
gpt_2_tokenizer.decode(tokens['input_ids'])
```

```
Out[33]: 'lationtersieny mind Lead ske theoret takezz comput Cl food terrorwards
Europe traditional Cl'
```

So essentially, using a different tokenizer will entirely confuse the model, as the token ids used would represent something entirely different. This would result in model not learning/inferring the correct thing.

Let's look at this from another angle. Not of the ids that are assigned to each token, but also how the sentence is converted into tokens.

```
In [34]: print('Tokens from BERT Tokenizer -', bert_tokenizer.tokenize(input_text))
print('Tokens from GPT2 Tokenizer -', gpt_2_tokenizer.tokenize(input_text))
```

```
Tokens from BERT Tokenizer - ['hello', ',', 'welcome', 'to', 'cs', '76', '##5
0', '-', 'natural', 'language', '.', 'we', 'hope', 'you', 'have', 'fun', '.']
Tokens from GPT2 Tokenizer - ['Hello', ',', 'Gwelcome', 'Gto', 'GCS', 'G76',
'50', 'G-', 'GNatural', 'GLanguage', '.', 'GWe', 'Ghope', 'Gyou', 'Ghave', 'G
fun', '.']
```

```
In [35]: print('Tokens IDs from BERT Tokenizer -', bert_tokenizer(input_text)['input_
print('Tokens IDs from GPT2 Tokenizer -', gpt_2_tokenizer(input_text)['input_
```

Tokens IDs from BERT Tokenizer - [101, 7592, 1010, 6160, 2000, 20116, 6146, 12376, 1011, 3019, 2653, 1012, 2057, 3246, 2017, 2031, 4569, 1012, 102]  
 Tokens IDs from GPT2 Tokenizer - [15496, 11, 7062, 284, 9429, 8684, 1120, 532, 12068, 15417, 13, 775, 2911, 345, 423, 1257, 13]

So, each tokenizer is different, and tightly coupled with its corresponding model as the model is trained on these indices. Therefore, using the correct tokenizer is important.

## Tensors, Batching, Padding, Truncation

```
In [36]: # Getting torch tensors just requires adding a parameter
tokens = bert_tokenizer(input_text, return_tensors='pt')
# Move to device
tokens = tokens.to(device=device) # 'cuda' for Nvidia GPUs
print('Type-', type(tokens['input_ids']), 'Device-', tokens['input_ids'].dev
```

Type- <class 'torch.Tensor'> Device- cuda:0

```
In [38]: # Can seamlessly handle batches and padding

input_batch = [
    "It does not do to dwell on dreams and forget to live.",
    "Yer a wizard, Harry.",
    "I solemnly swear that I am up to no good.",
    "After all this time? Always.",
    "Happiness can be found, even in the darkest of times, if one only remem
]

batch_tokens = bert_tokenizer(input_batch, return_tensors='pt', padding=True)
# Move to device
batch_tokens = batch_tokens.to(device=device) # 'cuda' for Nvidia GPUs
print('Type-', type(batch_tokens['input_ids']), 'Device-', tokens['input_ids
```

Type- <class 'torch.Tensor'> Device- cuda:0

```
In [39]: bert_tokenizer.batch_decode(batch_tokens['input_ids']) # pads to max length
```

```
Out[39]: ['[CLS] it does not do to dwell on dreams and forget to live. [SEP] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]',
 '[CLS] yer a wizard, harry. [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PA
D] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]',
 '[CLS] i solemnly swear that i am up to no good. [SEP] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]',
 '[CLS] after all this time? always. [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [P
AD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]',
 '[CLS] happiness can be found, even in the darkest of times, if one only r
emembers to turn on the light. [SEP]']
```

```
In [40]: batch_tokens = bert_tokenizer(input_batch, return_tensors='pt', padding=True)
# Move to device
batch_tokens = batch_tokens.to(device=device) # 'cuda' for Nvidia GPUs
bert_tokenizer.batch_decode(batch_tokens['input_ids'])
```

```
Out[40]: ['[CLS] it does not do to dwell on dreams and forget to live. [SEP] [PAD]',
          '[CLS] yer a wizard, harry. [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]',
          '[CLS] i solemnly swear that i am up to no good. [SEP] [PAD] [PAD] [PAD]',
          '[CLS] after all this time? always. [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]',
          '[CLS] happiness can be found, even in the darkest of times, if one [SE',
          'P]']
```

You can also build on top of existing tokenizers, and add your own tokens. Refer to the docs for more details.

## 2.2. Models

The Hugging Face transformers library provides access to a wide array of state-of-the-art Transformer models designed for NLP, Vision, Multimodal and other tasks. These models, including BERT, GPT, T5, and many others, are pre-trained on massive datasets and fine-tuned for specific tasks. The library offers a unified API to load, train, and deploy these models, making it easy for users to leverage advanced capabilities without needing to build models from scratch. The advantage of using transformers lies in its extensive model hub, where users can find and share pre-trained models, thus accelerating development and enabling consistent, high-quality performance across various NLP applications.

```
In [41]: from transformers import BertForSequenceClassification

bert_model = BertForSequenceClassification.from_pretrained("textattack/bert-
bert_model

pytorch_model.bin: 0%|          | 0.00/438M [00:00<?, ?B/s]
model.safetensors: 0%|          | 0.00/438M [00:00<?, ?B/s]
```

```

Out[41]: BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (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): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=
True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=Tr
ue)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)

```

`.from_pretrained()` method can also be used to import a local checkpoint from `cache_dir`.

This is now an already trained PyTorch model, everything else that PyTorch supports will be supported with this model, with only one line of code and no pre-training.

## Model Configuration

Every model has a config file. The model config is a vital component that encapsulates the architecture, behavior, and hyperparameters governing a model's operation. This configuration file includes details such as the number of hidden layers, attention heads, and the dimensionality of hidden states, which define the model's architecture. It also specifies hyperparameters like learning rate, dropout rate, and initialization settings, crucial for training and fine-tuning the model. Additionally, `model.config` manages task-specific parameters, such as the number of labels for classification tasks and whether to output attention weights or hidden states. Beyond the configuration, understanding Hugging Face models involves grasping key concepts like tokenization, fine-tuning, and the use of pre-trained models across different tasks.

```
In [42]: bert_model.config
```

```
Out[42]: BertConfig {
  "_attn_implementation_autoset": true,
  "_name_or_path": "textattack/bert-base-uncased-yelp-polarity",
  "architectures": [
    "BertForSequenceClassification"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "finetuning_task": "yelp_polarity",
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.48.0",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```

## Different Types of BERT (and other models) Available

BERT Docs: [https://huggingface.co/docs/transformers/v4.44.2/en/model\\_doc/bert](https://huggingface.co/docs/transformers/v4.44.2/en/model_doc/bert)

Generally, for all models, there will be many types of models available in

BertModel  
 BertForPreTraining  
 BertLMHeadModel  
 BertForMaskedLM  
 BertForNextSentencePrediction  
 BertForSequenceClassification  
 BertForMultipleChoice  
 BertForTokenClassification  
 BertForQuestionAnswering

HuggingFace. For example, BERT has the following -

Why?

The reason for having so many variations is that BERT is a versatile model that can be adapted to a wide range of NLP tasks. Each of these variations takes the core BERT architecture and tailors it for a specific task by adding task-specific heads (layers) on top of the base model. This specialization allows users to fine-tune BERT more effectively for the particular needs of their applications, ensuring better performance and more accurate results.

```
In [43]: from transformers import BertModel

bert_base = BertModel.from_pretrained("google-bert/bert-base-uncased", cache
bert_base
```

```
config.json: 0%|          | 0.00/570 [00:00<?, ?B/s]
model.safetensors: 0%|          | 0.00/440M [00:00<?, ?B/s]
```

```

Out[43]: BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
      (0-11): 12 x BertLayer(
        (attention): BertAttention(
          (self): BertSdpaSelfAttention(
            (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): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
          (intermediate_act_fn): GELUActivation()
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
)

```

You will find similar variants for GPT2.

Knowing your task, importing the correct model and using it correctly is important. Not doing this won't show any syntax errors, but your outputs/learning won't be correct.

### 3. Evaluate

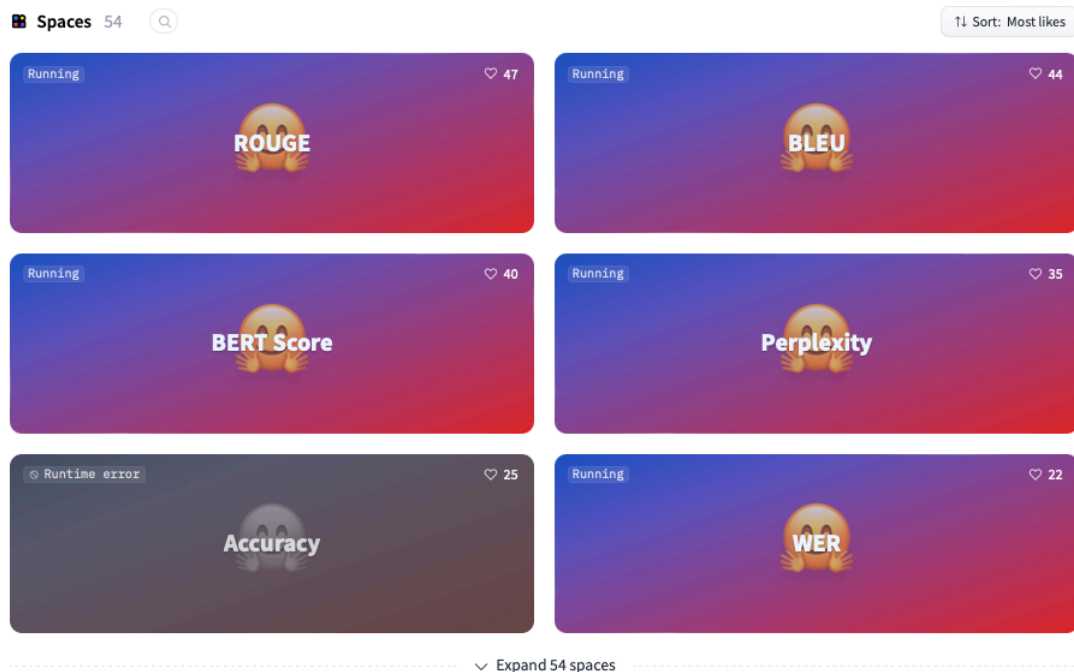
The Hugging Face evaluate library is a powerful tool designed to streamline the process of evaluating machine learning models, particularly in the context of natural



language processing (NLP). It provides a wide range of standardized evaluation metrics and datasets, allowing users to assess the performance of their models with ease. By abstracting the complexity of metric computation, the evaluate library enables quick integration into model development pipelines, ensuring consistent and reproducible evaluations across different models and tasks. One of its key advantages is its compatibility with the Hugging Face datasets and transformers libraries, making it simple to pair model predictions with appropriate metrics. This integration reduces the overhead of custom evaluation code, promotes best practices in model validation, and facilitates the comparison of models by providing access to a common set of metrics used in the community.

Docs: <https://huggingface.co/docs/evaluate>

Some supported metrics: <https://huggingface.co/evaluate-metric>



```
In [47]: from evaluate import load

rouge = load('rouge', cache_dir=cache_dir)
predictions = [
    "It does not do to dwell on dreams and forget to live.",
    "After all this time? Always."
]
references = [
    "It does not do well to dwell on dreams and forget to live.",
    "After all this time? Yes, always."
]
results = rouge.compute(predictions=predictions, references=references)
print(results)
```

Downloading builder script: 0.00B [00:00, ?B/s]

```
{'rouge1': 0.9345454545454546, 'rouge2': 0.7681159420289854, 'rougeL': 0.9345454545454546, 'rougeLsum': 0.9345454545454546}
```

Supports batches and aggregation by default.

This provides other metrics like accuracy, recall, f1-score as well. But where it excels from other libraries is its text generation metrics support, and seamless integration with other Huggingface libraries.

## 4. Examples

We shall now cover two examples -

1. Finetuning a DistilBERT for Text Classification
2. Text Generation using Gemma-2B (we shall only show inference, but you can also fine-tune it or use it in any other way like a PyTorch model)

### 4.1. Fine-tuning a DistilBERT for Text Classification

Here, we shall integrate all modules discussed above to fine-tune a DistilBERT for Sequence Classification.

DistilBERT is a streamlined, efficient version of the original BERT model, developed by Hugging Face to offer a balance between performance and computational efficiency. With 40% fewer parameters and a 60% faster inference time, DistilBERT achieves about 97% of BERT's performance across various NLP tasks, including text classification and question answering. It is trained using knowledge distillation, where the smaller model learns from the outputs and intermediate representations of a larger, pre-trained BERT model. By reducing the number of layers from 12 to 6 and eliminating the next sentence prediction objective, DistilBERT becomes a practical solution for deploying powerful NLP models in resource-constrained environments, such as mobile devices and real-time applications.

```
In [48]: from datasets import load_dataset
         from torch.utils.data import DataLoader
         from transformers import AutoModelForSequenceClassification, TrainingArguments
         from evaluate import load
```

```
In [49]: # The entire code of loading a dataset, creating data loader, creating a tok
dataset = load_dataset('ccdv/patent-classification', cache_dir=cache_dir)
```

```
In [50]: label_info = dataset['train'].features['label']
         id2label = {i: label for i, label in enumerate(label_info.names)}
         label2id = {label: i for i, label in enumerate(label_info.names)}
```

```
In [51]: distilbert_tokenizer = AutoTokenizer.from_pretrained("distilbert/distilbert-
         distilbert_model = AutoModelForSequenceClassification.from_pretrained("disti
```

```
tokenizer_config.json: 0%|          | 0.00/48.0 [00:00<?, ?B/s]
config.json: 0%|          | 0.00/483 [00:00<?, ?B/s]
vocab.txt: 0.00B [00:00, ?B/s]
tokenizer.json: 0.00B [00:00, ?B/s]
model.safetensors: 0%|          | 0.00/268M [00:00<?, ?B/s]
```

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert/distilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight', 'pre\_classifier.bias', 'pre\_classifier.weight']  
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

In [52]: `distilbert_model`

```
Out[52]: DistilBertForSequenceClassification(
  (distilbert): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0-5): 6 x TransformerBlock(
          (attention): DistilBertSdpaAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
    (pre_classifier): Linear(in_features=768, out_features=768, bias=True)
    (classifier): Linear(in_features=768, out_features=9, bias=True)
    (dropout): Dropout(p=0.2, inplace=False)
  )
```

Now, instead of Tokenizing every batch every epoch, we can use HuggingFace functions to do this seamlessly in one go for the entire dataset.

```
In [53]: def preprocess_function(examples):
          return distilbert_tokenizer(examples["text"], truncation=True)
          # since we are using the distilbert_tokenizer, this will automatically be tr
```

```
In [54]: tokenized_dataset = dataset.map(preprocess_function, batched=True)
```

```
Map:    0%|          | 0/25000 [00:00<?, ? examples/s]
Map:    0%|          | 0/5000 [00:00<?, ? examples/s]
Map:    0%|          | 0/5000 [00:00<?, ? examples/s]
```

```
In [55]: # Creating a subset only to reduce training time, since this is just for dem
          from torch.utils.data import Subset

          subset_train = Subset(tokenized_dataset['train'], list(range(5000)))
          subset_val = Subset(tokenized_dataset['validation'], list(range(500)))
```

We will now define a **Data collator**.

Data collators are utility classes provided in the transformers library that help prepare batches of data for training or evaluation in a model-friendly format. They handle tasks such as padding sequences to the same length within a batch, dynamically adjusting padding during training, and creating appropriate input formats for specific tasks, such as language modeling or sequence classification.

Docs:

[https://huggingface.co/docs/transformers/v4.44.2/en/main\\_classes/data\\_collator](https://huggingface.co/docs/transformers/v4.44.2/en/main_classes/data_collator)

```
In [56]: from transformers import DataCollatorWithPadding

          data_collator = DataCollatorWithPadding(tokenizer=distilbert_tokenizer)
```

```
In [57]: # Defining evaluation metric
          import numpy as np

          accuracy = load("accuracy", cache_dir=cache_dir)

          def accuracy_metric(eval_pred):
              predictions, labels = eval_pred
              predictions = np.argmax(predictions, axis=1)
              return accuracy.compute(predictions=predictions, references=labels)
```

Now let's start our training. We can also use the PyTorch training loop as discussed during the PyTorch tutorial (since these models are ultimately PyTorch models). But, let's utilize a more abstract training API provided by HuggingFace.

## Huggingface Trainer API

The Trainer API is a high-level abstraction that simplifies the process of training, evaluating, and fine-tuning transformer models. It handles many of the complexities of model training, such as data loading, optimization, gradient accumulation, mixed

precision training, and more. This document provides an overview of the key components and steps involved in using the Trainer API.

#### Key Components of the Trainer API:

- **Model:** The pre-trained model or custom model you want to fine-tune or train.
- **TrainingArguments:** A configuration class that specifies the parameters for training, such as learning rate, batch size, number of epochs, and more.
- **Datasets:** The training and evaluation datasets, which can be in the form of Hugging Face Datasets or PyTorch datasets.
- **Data Collator:** An optional component that handles data formatting and padding for batches.
- **Tokenizer:** The tokenizer associated with the model, used for preprocessing text data.
- **Compute Metrics:** A function for calculating evaluation metrics during validation.

Docs: [https://huggingface.co/docs/transformers/v4.44.2/en/main\\_classes/trainer](https://huggingface.co/docs/transformers/v4.44.2/en/main_classes/trainer)

#### Define Training arguments -

Set up the training configuration using TrainingArguments. This configuration class includes various parameters that control the training process, such as:

- **output\_dir:** Directory where the model checkpoints and logs will be saved.
- **learning\_rate:** The learning rate for the optimizer.
- **per\_device\_train\_batch\_size:** The batch size per GPU/TPU core/CPU for training.
- **per\_device\_eval\_batch\_size:** The batch size per GPU/TPU core/CPU for evaluation.
- **num\_train\_epochs:** The number of epochs to train the model.
- **weight\_decay:** The weight decay to apply to the optimizer.
- **eval\_strategy:** When to run evaluation (e.g., at each epoch).
- **save\_strategy:** When to save model checkpoints (e.g., at each epoch).
- **load\_best\_model\_at\_end:** Whether to load the best model found during training at the end.
- **push\_to\_hub:** Whether to push the model to the Hugging Face Model Hub. These arguments are crucial for defining how your model will be trained and evaluated.

```
In [58]: training_args = TrainingArguments(
    output_dir="./cache_dir/cs7650_tutorial_finetuning_classification",
    learning_rate=2e-5, # use less learning rate with fine-tuning
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True, # will automatically load the best model at
```

```

push_to_hub=False, # can push model to Huggingface hub
report_to='none' # can export results to visualizing tools like wandb
)

```

After setting up the TrainingArguments, create the Trainer object. The Trainer manages the entire training process and is initialized with the following components:

- Model: The pre-trained model or custom model you want to fine-tune or train.
- Arguments: The TrainingArguments that define the training configuration.
- Datasets: The training and evaluation datasets, which can be from the Hugging Face Datasets library.
- Tokenizer: The tokenizer used to preprocess the text data for the model.
- Data Collator (optional): A function or class that handles data formatting, such as padding sequences to the same length within a batch.
- Compute Metrics (optional): A function that calculates evaluation metrics during validation.

```

In [59]: trainer = Trainer(
    model=distilbert_model,
    args=training_args,
    train_dataset=subset_train, # truncating for demonstration
    eval_dataset=subset_val,
    tokenizer=distilbert_tokenizer,
    data_collator=data_collator,
    compute_metrics=accuracy_metric,
)

```

/tmp/ipykernel\_1235626/1288456583.py:1: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.\_\_init\_\_`. Use `processing\_class` instead.

```

trainer = Trainer(
WARNING:accelerate.utils.other:Detected kernel version 5.4.0, which is below the recommended minimum of 5.5.0; this can cause the process to hang. It is recommended to upgrade the kernel to the minimum version or higher.
)


```

Now we shall train the model, that only requires a single function call.

```

In [60]: trainer.train()

```

 [1875/1875 03:52, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy
1	1.476400	1.348720	0.534000
2	1.103100	1.272423	0.562000
3	0.944700	1.280792	0.560000

```

Out[60]: TrainOutput(global_step=1875, training_loss=1.109807958984375, metrics={'train_runtime': 233.6041, 'train_samples_per_second': 64.211, 'train_steps_per_second': 8.026, 'total_flos': 1987259028480000.0, 'train_loss': 1.109807958984375, 'epoch': 3.0})

```

## Model Inference

Go and check the folders and files created during training. We can pick any checkpoint from it to conduct our inferences.

```
In [61]: # Load the finetuned model from local
test_data = dataset['test']
finetuned_tokenizer = AutoTokenizer.from_pretrained("./cache_dir/cs7650_tuto
finetuned_model = AutoModelForSequenceClassification.from_pretrained("./cach
```

```
In [62]: # Collect some positive and negative cases
test_data = dataset['test']
positive_examples = []
negative_examples = []
limit_each = 5 # generate limit_each number of positive and negative example
finetuned_model = finetuned_model.to(device=device)
for i in range(0, len(test_data)):
    text = test_data[i]['text'] # input
    label_id = test_data[i]['label']
    tokens = finetuned_tokenizer(text, truncation=True, return_tensors = 'pt
    logits = finetuned_model(**tokens).logits # forward pass of model
    predicted_class_id = logits.argmax().item() # should do softmax ideally
    predicted_label = finetuned_model.config.id2label[predicted_class_id] #
    actual_label = finetuned_model.config.id2label[label_id]
    if label_id==predicted_class_id and len(positive_examples) < limit_each:
        positive_examples.append([text, predicted_label, actual_label])
    elif label_id!=predicted_class_id and len(negative_examples) < limit_eac
        negative_examples.append([text, predicted_label, actual_label])
    if len(positive_examples) >= limit_each and len(negative_examples) >= li
        break
```

```
In [63]: import pandas as pd
```

```
In [64]: pd.DataFrame(positive_examples, columns=['text', 'predicted class', 'true clas
```

```
Out[64]:
```

	text	predicted class	true class
0	an alarm network for a building 8 , or the lik...	Physics	Physics
1	the disposable razor and emollient dispensing ...	Performing Operations; Transporting	Performing Operations; Transporting
2	in accordance with the present invention it ha...	Chemistry; Metallurgy	Chemistry; Metallurgy
3	in the following , the invention will be descr...	Physics	Physics
4	detecting / diagnosing : the terms detecting a...	Human Necessities	Human Necessities

```
In [65]: pd.DataFrame(negative_examples, columns=['text', 'predicted class', 'true clas
```

Out [65]:

	text	predicted class	true class
0	as used herein , the term &#34 ; sensitizer m...	Chemistry; Metallurgy	General tagging of new or cross-sectional tech...
1	fig1 describes the five step / stage process o...	Chemistry; Metallurgy	Performing Operations; Transporting
2	an embodiment of the invention will be describ...	Physics	Electricity
3	in the discussion which follows , it is import...	Physics	Chemistry; Metallurgy
4	fig1 diagrammatically illustrates hydraulic ap...	Mechanical Engineering; Lightning; Heating; We...	General tagging of new or cross-sectional tech...

In [66]: *#Visualizing a negative example*  
negative\_examples[4][0]



Out[66]: 'fig1 diagrammatically illustrates hydraulic apparatus for controlling a plurality of valves or other subsea operators while using only a pair of hydraulic pressure source lines . the present invention as illustrated in fig1 includes a control module 11 for use with a subsea christmas tree 12 having a plurality of hydraulically controlled valves with their associated valve operators . the control module 11 is connected to a surface control center 13 having the usual pressure pumps , pressure gauges and switches , none of which is shown . the control module 11 includes a rotary actuator 17 connected to a rotary switch having a plurality of rotatable valve sections 18 - 24 each having a pressure input pa - pg , an output oa - og , and a vent va - vg . each valve section includes a plurality of positions a - f each having either the pressure input connected to the output or having the output connected to the vent . each of the pressure inputs pa - pg is connected to the hydraulic supply line 28 which is provided with pressurized fluid from the control center 13 , and each of the vents va - vg is connected to a vent 29 which is vented into the sea . an accumulator 30 , which is connected to the hydraulic supply line 28 , aids in providing a stabilized value of hydraulic pressure to the valve sections 18 - 24 and to operate the actuator 17 through a pilot valve 34 . the rotary actuator 17 includes a rotatable shaft 17a which is coupled to a plurality of rotatable shafts 18a - 24a of the valve sections 18 - 24 . when a hydraulic pilot line 35 is unpressurized the pilot valve 34 is in the position shown in fig1 wherein upper chamber 17b of the actuator 17 is connected to the vent 29 through the valve section a , whereby the actuator shaft 17a and the valve sections 18 - 24 remain in a stationary position . when pressure is admitted to the hydraulic pilot line 35 the spool of the valve 34 shifts so that liquid from the hydraulic supply line 28 is coupled through the section b of the pilot valve 34 to the upper chamber 17b , causing the actuator 17 to rotate the valve sections 18 - 24 to another distinct position . when the valves 18 - 24 are in the positions shown in fig1 hydraulic pressure from the hydraulic supply line 28 is coupled through portion a of valve 19 through a hydraulic line 25b to a production wing valve 36 , causing the wing valve to open . hydraulic fluid coupled through portion a of valve 20 and portion a of valve 23 through hydraulic lines 25c , 25f also causes a downhole safety valve 37 and a crossover valve 41 to open . an upper master valve 42 and a lower master valve 43 are connected to vent 29 through hydraulic line 25a and the portion a of valve 18 , and an annulus master valve 47 and an annulus wing valve 48 are connected to the vent 29 by lines 25d , 25e through the portions a of valve section 21 and 22 , respectively . the lowermost valve 24 and a plurality of pressure relief valves 51 - 56 provide a predetermined upper value of pressure on the pilot line 35 at the control center to indicate the position of the valve 24 , thereby also indicating the position of the rotary actuator 17 and of the other valve 18 - 23 . for example , when all the valves are in their a positions , the pressure relief valve 51 is connected through the portion a of valve 24 to the pilot line 35 and limits the maximum pressure on the pilot line to 1000 psi . when the valves are all in their b positions the pressure relief valve 52 limits the pressure on the pilot line 35 to 1400 psi , thereby indicating that the valves and the actuator are in said b positions . fig1 and 15 disclose other embodiments of circuits for interconnecting the actuator pilot valve 34 , the actuator 17 and the valve sections . in fig1 the position signal line 35a is connected to the supply line 28 through the portion b of the pilot valve 34 so that the pressure on line 28 can be used to determine the rotary position of the valve sections 18 - 24 during the time when the pilot valve 34 is energized . when the pilot valve 34 is deenergized the signal line 35a is connected to the vent 29 . the hydraulic supply line 28 is always connected to the valve section just as shown

n in fig1 . the hydraulic supply line 28 of fig1 is connected to a feed line 28a when the pilot valve 34 is deenergized . when the pilot line 35 of fig1 is energized the supply line 28 is disconnected from the feedline 28a thereby allowing the rotary position of the valves to be indicated by the pressure on line 28 and disconnecting the christmas tree valves until the pilot valve 34 is again deenergized . the valve sections 18 - 24 are individually removable units which can be stacked in an end - to - end manner as shown in fig2 with the shaft 17a of the actuator 17 connected to the shaft 18a of the uppermost valve section 18 , and with each of the other valve sections having a shaft connected to the shaft of the valve section positioned immediately above it . each of the valve sections includes a body 60 ( fig3 ) having a generally cylindrical chamber 61 , with a generally disc shaped rotor 62 mounted therein and a cover 63 connected to the body 60 by a plurality of cap screws 67 . the rotor 62 is rotatably mounted in the chamber 61 by the shaft 18a which is pressed into a bore 69 in the rotor ( fig3 ) . the rotor is connected to the shaft and retained in any one of six positions ( fig6 ) thereon by a key 70 ( fig3 ) which resides partially in a slot 74 in the shaft , and partially in any one of the six slots 75 in the rotor 62 . one end of the shaft 18a includes a female slot 76 ( fig5 ) , and the other shaft end includes a mating male projection 79 ( fig4 ) . when the valve sections are stacked , the male projection 79 on the upper end of one shaft fits into the female slot 76 on the lower end of an adjacent valve section to interconnect the sections for unitary rotation of their shafts and rotors . a plurality of ball or other suitable bearings 80 , mounted between the rotor 62 , the body and the cover 63 , relieve strain at the connection between the shaft and the rotor . the valve section ( fig3 ) includes a right angle pressure port 81 and a vent port 82 drilled radially in the body , and a right angle outlet port 86 in the cover 63 . a pair of shear seals 87a , 87b are mounted in an enlarged portion 81a , 86a of the ports 81 , 86 to provide a fluid - tight seal between the ports and the rotor . the rotor 62 ( fig3 ) includes a plurality of holes 88a - 88f drilled in either a straight - line pattern or a right angle pattern . when any one of the right angle holes 88b , 88d and 88f are rotated adjacent the shear seals 87a , 87b the outlet port 86 is connected to the vent port 82 as shown in fig3 . when any one of the straight line holes 88a , 88c and 88e are rotated adjacent the shear seals the pressure port 81 is connected directly to the outlet port 86 . the drilling pattern of the rotor shown in fig6 is shown schematically in fig7 . a plurality of threaded mounting bores 92 ( fig4 ) and a plurality of dowel pin holes 93 facilitate mounting the valve sections to a manifold 94 ( fig2 ) . a plurality of seals 98 provide sealing around the shaft 18a , and a plurality of seals 99 provide fluid tight connections between the pressure ports 81 , the vent port 82 , the outlet port 86 and the connections on the manifold 94 . a seal 101 provides a fluid tight connection between the body 60 and the cover 63 . in order to control a plurality of subsea valves in a desired sequence it may be necessary to provide a unique pattern of straight - through and right angle holes for the rotors of each of the valve sections . for example , to control the subsea tree 12 of fig1 the sequence of operation shown in the matrix of fig1 may be used . in the shut - in position , with the supply line 28 and the pilot line 35 unpressurized , the crossover valve 41 ( fig1 ) is open , as represented by the letter o ( fig1 ) , and all the other valves are closed , as represented by the letter c . in the example shown in fig1 , the valves 42 , 43 are closed in steps 0 , 1 and 3 , and are open in steps 2 and 4 - 6 . this requires a rotor with the pattern of straight - through and angular holes shown in type no . 11 of fig8 b , and with the rotor starting in the f position in order to provide pressure to the valves 42 , 43 in steps 2 and 4 - 6 . this can be done by

positioning and locking the rotor 62 on the shaft 18a ( fig3 ), with the right angle hole f at the shaft &# 39 ; s no . 1 position . a type 10 valve rotor ( fig8 b ) with the right angle hole e at the no . 1 shaft position , is required to control the valve 36 . proper positioning of the twelve rotor types of fig8 a and 8b with respect to the actuator drive shaft facilitates operation of the tree 12 in a six - step manner , and also all possible combinations of sequences of operations of the valves . other combinations , and / or specially drilled rotors , are not required when the twelve basic rotor types shown in fig8 a and 8b are used . in a system where a different number of steps of operation are required , a different number of holes can be spaced around the rotor , and the rotors rotated a different number of degrees as each step of the operation is carried out . the manifold 94 ( fig2 ) is mounted on a base plate 104 , and includes a plurality of internal passageways ( not shown ) which connect the ports 81 , 82 , 86 of the valve sections 18 - 23 to the various hydraulic lines , such as the supply line 28 , pilot line 35 and output lines 25a - 25f of fig1 and connect the pressure relief valves 51 - 56 to the valve section 24 . the valve sections 18 - 24 ( fig2 ) are secured to the manifold 94 by a plurality of cap screws 100 which are threaded into bores 92 ( fig4 ) in the valve bodies 60 . the actuator 17 is fixed to the upper end of the manifold 94 . a housing 103 , sealed to the base plate 104 by a seal 108 , provides fluid - tight protection to the actuator 17 and valve assembly . the actuator 17 ( fig9 - 12 ) includes a generally cylindrical tubular housing 105 having a pair of end plates 106 , 107 connected thereto by a plurality of cap screws 111 . the upper end plate 106 includes an axially extending fluid chamber 112 and a right angle port 113 extending between the upper end ( fig9 ) of the fluid chamber and a hydraulic line 109 . the lower end plate 107 includes an axial flange 117 and an axial bore 118 extending through the center of the end plate . a spring loaded detent 123 , having a ball - shaped portion 124 at the radial inner end thereof , resides in a radial hole 119 . the cylindrical actuator shaft 17a includes a lower portion 17c mounted in the axial bore 118 . a pair of roller bearing assemblies 125 , 126 , mounted in a pair of recesses 130 , 131 of the lower end plate 107 , rotatably mount the shaft to the lower end plate . the lower portion of the actuator shaft is threaded to a nut 132 . the shaft 17a includes an enlarged upper portion 17d having an axially extending bore 136 . an axially movable plunger 137 ( fig9 ) is located in the upper end of the housing 105 , and this plunger includes a piston 138 extending upwardly into the fluid chamber 112 of the upper end plate 106 , and a radial flange 137a extending to the wall of the housing 105 . the lower portion of the plunger 137 includes a sleeve 139 having a radially expanded portion 139a . a cylindrical pin 143 , having a pair of circumferentially extending grooves 144 , 145 , is mounted in a radial bore 149 in the expanded sleeve 139a . a spring detent 150 ( fig9 ), mounted in a radial bore 151 ( fig9 ), intersects the pin 143 and rests in one of the grooves 144 , 145 . a key 155 ( fig9 ), connected to the plunger 137 and riding in an axial slot 156 in the housing 105 , prevents rotational movement of the plunger 137 but allows vertical movement thereof relative to the housing . a coil spring 157 ( fig9 ), connected between the lower end plate 107 and the radial flange 137a , biases the plunger in an upward direction , so that in the absence of hydraulic pressure on the upper end of the piston 138 the plunger flange 137a moves upward to rest against the upper end plate 106 . a hollow cylindrical outer cam 161 ( fig9 ), having six angular slots 162a - 162e ( only five shown ) spaced about the upper portion thereof , is rotatably mounted around the upper portion 17c of the actuator shaft 17a by a plurality of bearings 160 . the outer cam also includes an additional angled slot which is not shown due to the problem of cluttering the drawings with t

oo many details . a plate 163 , having an axial bore 164 , is mounted in an annular groove 168 ( fig9 ) in the shaft 17a and secured to the outer cam by a plurality of cap screws 169 . the lower portion of the diagrammatic drawing of fig1 has been stretched to better show other details , so the groove 168 in fig1 appears to be much wider than the same groove as shown in the sectional view of fig9 . a torsion spring 170 ( fig9 ) is connected between the shaft 17a and the outer cam 161 to bias a radial inward cam lug 174 ( fig1 , 11 ) toward a radial outward shaft lug 175 . the rotation of outer cam 161 about the shaft 17a is limited to an arc of less than 360 degrees by the lugs 174 , 175 . a cylindrical inner cam 176 ( fig9 , 12 ) , having a plurality of angular slots 180a - 180f spaced about the upper portion ( fig1 ) thereof , is rotatably mounted in the axial bore 136 of the shaft 17a by a plurality of bearings 181 . the slots 180a - 180f ( fig1 ) are angled clockwise as they extend downward from the top of the inner cam 176 , in contrast to the slots 162a - 162e of the outer cam 161 which are angled counterclockwise as they extend downward . a torsion spring 182 ( fig9 ) is connected between the shaft 17a and the inner cam 176 to bias a radial outward cam lug 186 ( fig1 , 12 ) toward a radial inward shaft lug 187 . the actuator shaft 17a is normally retained in one of six rotary positions by the detent 123 ( fig9 ) extending into one of a plurality of shallow bores 188 in the lower portion 17c of the shaft 17a . as stated above , the number of rotary positions of the shaft and of the valve sections can be changed to a greater or lesser number as required . when the fluid chamber 112 ( fig9 ) is unpressurized the plunger 137 is biased to the upper end of the housing 105 with the pin 143 slightly above the upper end 161a of the outer cam 161 , and with one end of the pin 143 ( fig1 ) radially above the open end of one of the outer cam slots 162a - 162e and the other end of the pin 143 radially above the open end of one of the inner cam slots 180a - 180f . when the pin or cam follower 143 ( fig1 ) is positioned radially outward with the outward end 143a immediately above the open end of one of the outer cam slots 162a - 162e , the detent 150 is positioned in the groove 144 to retain the pin in the outer cam slot . when pressurized fluid is admitted through the port 113 to the fluid chamber 112 , the plunger 137 ( fig9 ) is moved axially downward forcing the pin 143 downward in the adjacent slot , with the pin moving along the radially extending line a ( fig1 ) , as the plunger is prevented from rotating by the key 155 in the slot 156 ( fig9 ) . for example , when the pin 143 moves down into the slot 162e , the pin 143 progresses downward along the line a until it reaches the lower end 162e of the slot causing the outer cam 161 to rotate 60 degrees clockwise ( as viewed from above the actuator ) . clockwise rotation of the cam 161 and the cam lug 174 causes the shaft lug 175 ( fig1 ) and the shaft 17a to rotate 60 degrees clockwise and for the detent 123 to move from bore 188a ( fig1 ) into the adjacent detent bore 188b . when pressure is released from the chamber 112 ( fig9 ) the plunger 137 is forced upward by the coil spring 157 , moving the pin 143 ( fig1 ) upward along the line a , in the slot 162e and rotating the outer cam 60 degrees counterclockwise . the shaft 17a is prevented from rotating by the detent 123 in the bore 188b and the cam lug 174 ( fig1 ) is rotated counterclockwise away from shaft lug 175 . the counterclockwise rotation of the outer cam 161 winds the torsion spring 182 ( fig9 ) to bias the outer cam lug 174 more strongly toward the shaft lug 175 . when the pin 143 moves out of the open end of the slot 162e the spring 170 causes the cam 161 to quickly rotate 60 degrees clockwise , with the upper end of the cam slot 162f stopping adjacent the pin 143 , as the outer cam lug 174 contacts the shaft lug 175 . this same sequence is repeated each time the plunger moves down and returns to the upper position , with the outer cam and shaft rotating 60 degrees clockwise as the pin 143 moves down to

the bottom of a cam slot , the shaft is stopped and held in place by the detent 123 while the pin moves upward , rotates the outer cam counterclockwise and winds up the tension spring . when the pin 143 moves above the outer cam the cam snaps around clockwise with the pin adjacent another open end of another slot . this causes the shaft to rotate in increments between six distinct stopping points . when the pin 143 reaches the lower end of the last cam slot 162f the radial outer end of the pin is pressed against a bevelled surface 192 ( fig9 ) causing the pin 143 to move radially inward with the end 143b of the pin in the lower end 180a &# 39 ; of the cam slot 180a in the inner cam 176 . the detent 150 moves into the groove 145 of the pin to retain the pin in the slot 180a as the pin 143 and the plunger 137 move upward . each time the plunger moves downward the pin 143 moves downward in one of the inner cam slots 180a - 180f , with the pin moving along a line parallel to line a ( fig1 ) causing the inner cam 176 to rotate counterclockwise 60 degrees . the radial outward lug 186 on the inner cam 176 presses against the radial inward lug 187 on the shaft 17a ( fig1 ) causing the shaft to rotate counterclockwise and to move the detent 123 ( fig1 ) into an adjacent bore 188a - 188 f ( only part of which are shown ) on the shaft 17a . an upward movement of the plunger 137 and the pin 143 causes the inner cam 176 ( fig1 ) to rotate clockwise to move the lug 186 away from the lug 187 and &# 34 ; wind &# 34 ; the torsion spring 170 ( fig1 , 12 ) to bias the inner cam lug 186 more strongly toward the shaft lug 187 . when the pin 143 moves upward , out of the open end of one of the slots 180a - 180f , the torsion spring 182 causes the cam 176 to quickly rotate 60 degrees counterclockwise with the upper end of the next cam slot stopping adjacent the pin 143 , as the inner cam lug 186 contacts the shaft lug 187 . repeating the sequence causes the shaft 17a to rotate counterclockwise in increments between each of six distinct stopping points . when the pin 143 reaches the lower end of the last cam slot 180f ( fig9 ) the radial inner end 143b of the pin is pressed against a bevelled surface 193 causing the pin 143 to move radially outward into the lower end of the cam slot 162a and to again reverse the direction of rotation of the actuator shaft whenever the plunger 137 is moved downward . the shaft rotates through six positions in one direction , then automatically reverses the direction of rotation , and moves in reverse order through the same six positions . this process is automatically repeated as long as the plunger moves down and back up . the actuator 17 is connected to the stacked valve sections 18 - 24 by a spring loaded coupling means 194 ( fig2 , 9 ) comprising a generally cylindrical coupler 198 having an axial bore 199 extending downward through a portion of the coupler . a radial flange portion 200 includes an annular groove 201 ( fig3 ) with one end of a compression spring 205 mounted in the groove 201 and the other end of the spring connected to an annular groove 206 in the nut 132 ( fig9 ) to bias the coupler 198 toward the valve section 18 ( fig2 ) . the coupler 198 is secured to the lower end 17c of the actuator shaft 17a ( fig9 ) by a set screw 207 mounted in a radial threaded hole 211 in the coupler and with the radial inner end of the set screw extending into an axial slot 212 in the shaft 17a . the slot 212 and set screw 207 allow the coupler 198 to travel axially along the shaft 17a through a distance determined by the vertical length of the slot 212 . the lower end of the coupler includes a shaft 213 ( fig3 ) having a radial slot 213a to receive the rectangular projection 79 on the upper end of the valve shaft 68 ( fig3 ) and to secure the shaft 18a of the upper valve section 18 to the shafts 213 and 17a . a slot 217 in the upper portion of the coupler 198 and a slot 218 in the lower end of the actuator shaft 17a ( fig9 ) contain a key 219 which couple rotational motion from the shaft 17a to the coupler 198 . the purpose of the coupling means 194 is to couple the actuator shaft 17a to the shaft 18a of the valve section

s 18 - 24 during normal operation of the actuator 17 . if the actuator should fail , the coupler 198 ( fig2 ) can be pried upward from the valve section 18 by an appropriate tool until the shaft 213 ( fig3 ) of the coupler is disconnected from the shaft 18a of the upper valve section . a nut 223 ( fig2 ) extending from the lower end of the valve 24 and connected to the valve shaft 24a can be turned by an appropriate wrench to rotate the shaft 24a and temporarily operate the valves 18 - 24 . another embodiment 218 of the valve sections 18 - 24 ( fig2 ) is disclosed in fig1 with most of the elements functioning in a manner similar to the embodiment shown in fig3 - 6 . however , a rotor 262 and a shaft 268 are fixed together and the rotor cannot be removed and positioned in a different rotary position on the shaft as in the valve section of fig3 . the rotor 262 is supported by the shaft 268 which rotates in a bore 269 in the valve body 260 and in a bore 270 in a cover 263 . the upper end of the shaft 262 includes a square hole 270 and a square shaft end 271 on the other end . several of the valve sections 218 can be stacked with the square end 271 of one valve shaft fitting into the square hole 270 in the shaft of an adjacent valve section . these stacked valve sections 218 can be connected to a single actuator as shown in fig2 but individual sections cannot be removed or replaced without disconnecting the sections mounted below the section being removed . the present invention discloses apparatus for remote control of a relatively large number of hydraulically - operated subsea operators using only two hydraulic lines between a surface control center and a subsea device containing the operators . a valve actuator having a rotatable shaft or other output member which is movable to a plurality of distinct operating positions is coupled to a plurality of valves which are used to control the subsea operators . the valve actuator includes means for automatically reversing the direction of movement , so the valve is moved in a first direction through a sequence of distinct positions and then moved in a reverse direction through the same distinct positions . although the best mode contemplated for carrying out the present invention has been herein shown and described , it will be apparent that modification and variation may be made without departing from what is regarded to be the subject matter of the invention as defined in the appended claims . '

Some reasons why the model didn't perform very well (knowingly introduced to discuss a few points) -

1. The documents are very large, while distilbert only supports max tokens of 512. So, only the first 512 tokens of a text are considered to make the inference. Therefore, it is important to select a model that supports the number of tokens you expect to have in the input text. Usually available in the model config.

How to check the max sequence length supported by the model? - Check cell below.

2. We saw that some overfitting started happening in third epoch itself with a very low learning rate. This is because the model was getting fine-tuned, and not trained from scratch.

How to solve this? - Using larger dataset should further reduce overfitting and increase performance.

In [67]: `finetuned_model.config`

```

Out[67]: DistilBertConfig {
  "_attn_implementation_autoset": true,
  "_name_or_path": "./cache_dir/cs7650_tutorial_finetuning_classification/c
heckpoint-1250",
  "activation": "gelu",
  "architectures": [
    "DistilBertForSequenceClassification"
  ],
  "attention_dropout": 0.1,
  "dim": 768,
  "dropout": 0.1,
  "hidden_dim": 3072,
  "id2label": {
    "0": "Human Necessities",
    "1": "Performing Operations; Transporting",
    "2": "Chemistry; Metallurgy",
    "3": "Textiles; Paper",
    "4": "Fixed Constructions",
    "5": "Mechanical Engineering; Lightning; Heating; Weapons; Blasting",
    "6": "Physics",
    "7": "Electricity",
    "8": "General tagging of new or cross-sectional technology"
  },
  "initializer_range": 0.02,
  "label2id": {
    "Chemistry; Metallurgy": 2,
    "Electricity": 7,
    "Fixed Constructions": 4,
    "General tagging of new or cross-sectional technology": 8,
    "Human Necessities": 0,
    "Mechanical Engineering; Lightning; Heating; Weapons; Blasting": 5,
    "Performing Operations; Transporting": 1,
    "Physics": 6,
    "Textiles; Paper": 3
  },
  "max_position_embeddings": 512,
  "model_type": "distilbert",
  "n_heads": 12,
  "n_layers": 6,
  "pad_token_id": 0,
  "problem_type": "single_label_classification",
  "qa_dropout": 0.1,
  "seq_classif_dropout": 0.2,
  "sinusoidal_pos_embs": false,
  "tie_weights_": true,
  "torch_dtype": "float32",
  "transformers_version": "4.48.0",
  "vocab_size": 30522
}

```

## 4.2. Text Generation using SmoLLM-1.7B

Model Card: <https://huggingface.co/HuggingFaceTB/SmolLM-1.7B>

SmolLM is a 1.7 billion parameter language model tailored to deliver robust natural language processing capabilities in a compact form factor. Despite its smaller size compared to larger models, SmolLM excels in tasks such as text generation, completion, translation, and summarization, thanks to its efficient architecture. The model leverages advanced transformer-based techniques to optimize performance, enabling it to understand and generate coherent and contextually relevant text across various domains.

```
In [68]: from transformers import AutoTokenizer, AutoModelForCausalLM

smollm_tokenizer = AutoTokenizer.from_pretrained("HuggingFaceTB/SmolLM-1.7B")
smollm_model = AutoModelForCausalLM.from_pretrained("HuggingFaceTB/SmolLM-1.7B")

tokenizer_config.json: 0.00B [00:00, ?B/s]
vocab.json: 0.00B [00:00, ?B/s]
merges.txt: 0.00B [00:00, ?B/s]
tokenizer.json: 0.00B [00:00, ?B/s]
special_tokens_map.json: 0%|          | 0.00/831 [00:00<?, ?B/s]
config.json: 0%|          | 0.00/698 [00:00<?, ?B/s]
model.safetensors.index.json: 0.00B [00:00, ?B/s]
Downloading shards: 0%|          | 0/2 [00:00<?, ?it/s]
model-00001-of-00002.safetensors: 0%|          | 0.00/5.00G [00:00<?, ?B/s]
model-00002-of-00002.safetensors: 0%|          | 0.00/1.85G [00:00<?, ?B/s]
Loading checkpoint shards: 0%|          | 0/2 [00:00<?, ?it/s]
generation_config.json: 0%|          | 0.00/111 [00:00<?, ?B/s]

In [69]: input_text = "Natural Language Processing is"
inputs = smollm_tokenizer(input_text, return_tensors="pt").to(device=device)

In [70]: outputs = smollm_model.generate(**inputs, max_length=200) # max length speci
print(smollm_tokenizer.decode(outputs[0]))
```

Setting `pad\_token\_id` to `eos\_token\_id`:0 for open-end generation.

Natural Language Processing is a branch of Artificial Intelligence that deals with the interaction between computers and human language. It involves the development of algorithms and models that enable computers to understand, interpret, and generate human language. NLP has a wide range of applications, including speech recognition, machine translation, sentiment analysis, and text summarization.

One of the key challenges in NLP is the ambiguity of human language. Words can have multiple meanings, and sentences can be ambiguous, making it difficult for computers to understand and interpret them accurately. To overcome this challenge, NLP algorithms use techniques such as tokenization, part-of-speech tagging, and named entity recognition to break down sentences into smaller components and identify the meaning of each component.

Another challenge in NLP is the vast amount of data available. Natural language is a complex and dynamic field, and there is a lot of data available that can be used to train NLP algorithms. However, this data is often unstructured and difficult to process, making it challenging to extract meaningful

**Generated Text -**



Natural Language Processing is a branch of Artificial Intelligence that deals with the interaction between computers and human language. It involves the development of algorithms and models that enable computers to understand, interpret, and generate human language. NLP has a wide range of applications, including speech recognition, machine translation, sentiment analysis, and text summarization. One of the key challenges in NLP is the ambiguity of human language. Words can have multiple meanings, and sentences can be ambiguous, making it difficult for computers to understand and interpret them accurately. To overcome this challenge, NLP algorithms use techniques such as tokenization, part-of-speech tagging, and named entity recognition to break down sentences into smaller components and identify the meaning of each component. Another challenge in NLP is the vast amount of data available. Natural language is a complex and dynamic field, and there is a lot of data available that can be used to train NLP algorithms. However, this data is often unstructured and difficult to process, making it challenging to extract meaningful insights from it. To address this challenge, NLP algorithms use techniques such as machine learning and deep learning to process and analyze large amounts of data. One of the most exciting applications of NLP is in the field of chatbots. Chatbots are computer programs that can simulate human conversation and interact with users in a natural and intuitive way. They are used in a variety of applications, including customer service, e-commerce, and entertainment. Chatbots use NLP algorithms to understand and interpret user input, and to generate appropriate responses. Another application of NLP is in the field of sentiment analysis. Sentiment analysis is the process of analyzing text data to determine the sentiment or emotion expressed in the text. This can be used to analyze social media posts, customer reviews, and other forms of text data to gain insights into public opinion and sentiment. In conclusion, Natural Language Processing is a rapidly evolving field that has the potential to revolutionize the way we interact with computers and each other. With the increasing amount of data available and the growing demand for natural language processing applications, NLP is poised to become an essential tool for businesses and organizations in the years to come.<|endoftext|>

## Sampling of Outputs

Sampling of outputs refers to the process by which a language model generates text based on the probabilities of the next possible tokens (words, characters, etc.). Instead of always choosing the most likely next token (as in greedy decoding), sampling introduces randomness, allowing for more diverse and creative text generation.

Reference: <https://huggingface.co/blog/how-to-generate>

```
In [71]: # Top-k Sampling
outputs = smollm_model.generate(**inputs, max_length=500, do_sample=True, to
print(smollm_tokenizer.decode(outputs[0]))
```

Setting ``pad_token_id`` to ``eos_token_id`:0` for open-end generation.

Natural Language Processing is the field of AI research concerned with making computers "intelligent" enough to understand natural language, such as English, French, or even Chinese. This is a complex task due to multiple issues, such as the wide variation and ambiguity found in human language, and how these variations may affect the results of different applications. However recent advancements in AI have allowed for progress in this field, with applications such as machine translation, sentiment analysis, and text completion.

As technology continues to advance and evolve, so too will the capabilities of AI in this field. This creates a unique challenge for researchers, as they must continually adapt to changing language usage and patterns in order to ensure the accuracy and effectiveness of their models.

There are various techniques that researchers may use to tackle these challenges, such as deep learning, which involves using neural networks to mimic the human brain's ability to process and understand language. Another technique is to use statistical models, which involve analyzing large amounts of data to identify patterns and relationships between words and phrases.

Despite the challenges, the potential of NLP is immense, with applications that can have a significant impact on a wide range of industries, such as customer service, healthcare, finance, and more. As AI continues to advance, we can expect to see even more significant breakthroughs in this field, with the ability to process and understand natural language at an increasing level of sophistication.<|endoftext|>

### **Generated Text -**

Natural Language Processing is a fascinating and rapidly evolving field of artificial intelligence that enables computers to process and understand human language. In this blog post, we will provide an overview of some of the key concepts and techniques used in NLP. NLP combines techniques and theories from various fields, including computational linguistics, machine learning, statistics, and more. It involves working with natural human language in many forms, such as text, speech, and sign language. NLP is used in a wide range of applications, from search engines and email filtering to language translation and sentiment analysis. It is used in many industries, including healthcare, finance, education, and entertainment. One of the key challenges in NLP is understanding the nuances and complexities of human language, including idioms, metaphors, slang, and cultural differences. NLP algorithms must be able to account for these complexities and accurately interpret the intended meaning of a sentence. There are several techniques used in NLP, including tokenization, stemming, lemmatization, and part-of-speech tagging. Each technique helps to break down text into smaller units and analyze their meaning. Tokenization breaks text into individual words or tokens, while stemming and lemmatization convert words to their base or root form. Part-of-speech tagging analyzes the syntactic role of words in a sentence. Another important technique in NLP is machine learning, which involves training algorithms on large datasets of text and using statistical models to predict the meaning of new, unseen text. Machine learning models can analyze patterns and relationships in text to make accurate predictions or identify language-related information. NLP also involves a wide range of technologies and tools, including text mining, natural language generation, sentiment analysis, and more. In conclusion, NLP

is a powerful and versatile field that enables computers to understand and interact with human language in many ways. As the field continues to evolve, researchers and developers will continue to explore new methods and techniques to improve the accuracy and effectiveness of NLP algorithms. The field of Natural Language Processing (NLP) is rapidly evolving and transforming the way humans interact with technology. NLP can be defined as the science or art of creating computer systems that can understand, interpret, and generate human language. With the advancement in technology, NLP has become a vital component of many real-life applications, such as virtual assistants, chatbots, and machine translation.

```
In [72]: # Top-p Sampling
         outputs = smollm_model.generate(**inputs, max_length=500, do_sample=True, to
         print(smollm_tokenizer.decode(outputs[0]))
```

Setting `pad\_token\_id` to `eos\_token\_id`:0 for open-end generation.

Natural Language Processing is a branch of artificial intelligence that deals with the interactions between computers and human language. It involves the development of algorithms and models that enable computers to understand, interpret, and generate human language. NLP has become increasingly important in recent years due to the explosion of digital data and the need to make sense of it.

### ### Text Preprocessing

Before we can analyze text data, we need to preprocess it. Text preprocessing involves cleaning and transforming raw text into a format that can be easily analyzed. This includes tasks such as tokenization, stemming, and stopword removal.

Tokenization is the process of breaking down text into individual words or tokens. This is an important step because it allows us to analyze the individual words in our text data. For example, consider the following sentence:

```
...
This is a sentence.
...
```

After tokenization, we would have:

```
...
[This, is, a, sentence]
...
```

Stemming is the process of reducing words to their root form. This is useful for reducing the number of unique words in our text data, which can help improve the performance of some NLP algorithms. For example, the words "running", "runs", and "ran" can all be reduced to the root word "run".

Stopword removal is the process of removing common words that do not carry much meaning, such as "the", "and", and "a". These words are often referred to as "stopwords" and can be removed without significantly impacting the meaning of the text.

### ### Text Classification

Once we have preprocessed our text data, we can begin to analyze it. One common task in NLP is text classification, which involves assigning a category or label to a piece of text. For example, we might want to classify movie reviews as positive or negative.

To perform text classification, we can use a technique called logistic regression. Logistic regression is a linear model that is used for binary classification problems. It works by finding the best-fitting line that separates the two classes in our data.

Here is an example of how we might use logistic regression to classify movie reviews:

```
...
import numpy as np
from sklearn.linear_model import LogisticRegression
```

```
# Assume we have a dataset with movie reviews as features and labels  
X = np.array([[1, 2, 3], [4,
```

### Generated Text -

Natural Language Processing is a branch of artificial intelligence (AI) that enables computers to understand, interpret, and generate human language. The primary goal of NLP is to build systems that can comprehend and produce human language in a way that is both natural and useful. The main challenges in NLP include:

- **Ambiguity:** Human language is inherently ambiguous, with words and phrases often having multiple meanings. For example, the word "bank" can refer to a financial institution or the side of a river.
- **Context:** Understanding the context in which words and phrases are used is crucial for accurate interpretation. Consider the phrase "I saw her duck." Without context, the meaning is unclear.
- **Language Variation:** Different languages have unique grammar, syntax, and vocabulary. Adapting NLP systems to work across multiple languages can be complex.
- **Emotion and Tone:** Human language can convey emotions and tone through words and punctuation. Recognizing and interpreting these nuances is essential for effective communication.
- **Domain-Specific Language:** NLP systems often need to work with specialized languages and terminology specific to a particular field or industry.
- **Handling Inconsistencies:** Natural language is prone to inconsistencies, such as homonyms (words that sound the same but have different meanings) and idiomatic expressions.
- **Data Privacy and Ethical Concerns:** NLP systems may need to process sensitive or confidential information, raising concerns about data privacy and ethical use of language data.
- **Continuous Learning:** Language is constantly evolving, and NLP systems must adapt to new words, expressions, and slang to maintain relevance.
- **Understanding and generating human-like text:** This involves tasks like text summarization, sentiment analysis, and machine translation.
- **Speech Recognition:** NLP enables the conversion of spoken language into text, which is essential for voice-activated devices and transcription services.
- **Question-Answering Systems:** NLP can be used to create systems that can answer questions posed in natural language, such as those found in search engines.
- **Chatbots and Virtual Assistants:** NLP powers chatbots and virtual assistants that can understand and respond to user queries in a conversational manner.
- **Text Classification:** NLP can be used to categorize text into predefined categories, such as spam detection in email filtering.

- **Sentiment Analysis:** NLP is employed to analyze and understand the sentiment expressed in text, which is valuable for market research and social media monitoring.
- **Text Summarization:** NLP can automatically generate concise summaries of long documents or articles

Things to note - The generation completes when either max\_tokens number of tokens have been generated or EOS token is encountered.

## 5. Text Embeddings and Semantic Representations

Text embeddings are fundamental to modern NLP systems, providing a way to represent textual data as dense numerical vectors that capture semantic meaning. In this section, we'll explore different types of embeddings and learn how to use SentenceTransformers for creating high-quality text representations.

### 5.1. What are Text Embeddings?

**Embedding** is the general term for mapping text to a high-dimensional vector space where similar texts are positioned closer together. There are different types of embeddings depending on the granularity of text being embedded:

1. **Token Embeddings:** Send individual tokens (words, subwords, characters) to a high-dimensional vector space. These are typically the first layer in many neural network-based NLP systems and form the foundation for more complex representations.
2. **Sentence Embeddings:** Send entire sentences, paragraphs, or documents to a high-dimensional vector space. These capture the overall semantic meaning of the text and are particularly useful for tasks like semantic search, clustering, and similarity comparison.
3. **Contextual Embeddings:** Unlike static embeddings, these change based on the context in which a word appears. Models like BERT produce contextual embeddings where the same word can have different representations depending on its surrounding words.

The key advantage of embeddings is that they transform discrete text into continuous numerical representations that machine learning models can process effectively, while preserving semantic relationships between different pieces of text.

### 5.2. Using SentenceTransformers for Pre-trained Embeddings

SentenceTransformers is a Python framework that provides an easy method to compute dense vector representations for sentences, paragraphs, and documents. It's built on top of PyTorch and Transformers and offers pre-trained models optimized specifically for generating high-quality sentence embeddings.

Docs: <https://www.sbert.net/>

Let's start by installing and importing the necessary libraries:

```
In [73]: # Install sentence-transformers if not already installed
# !pip install sentence-transformers
```

```
In [74]: from sentence_transformers import SentenceTransformer
import numpy as np
```

We'll be using the `all-MiniLM-L6-v2` model, which provides a good balance of performance and efficiency for sentence embeddings. This model is well-regarded on the [MTEB](https://huggingface.co/spaces/mteb/leaderboard) (Massive Text Embedding Benchmark) leaderboard, which evaluates text embedding models across diverse tasks and datasets. MTEB serves as a comprehensive evaluation framework that helps researchers and practitioners compare different embedding models and select the most appropriate one for their specific use case. You can explore the full leaderboard and model comparisons at <https://huggingface.co/spaces/mteb/leaderboard> to understand how different models perform across various NLP tasks.

```
In [75]: # Load a pre-trained sentence transformer model
# We'll use 'all-MiniLM-L6-v2' which provides good performance with relative
model = SentenceTransformer('all-MiniLM-L6-v2', cache_folder=cache_dir)
```

```
modules.json: 0%|          | 0.00/349 [00:00<?, ?B/s]
config_sentence_transformers.json: 0%|          | 0.00/116 [00:00<?, ?B/s]
README.md: 0.00B [00:00, ?B/s]
sentence_bert_config.json: 0%|          | 0.00/53.0 [00:00<?, ?B/s]
config.json: 0%|          | 0.00/612 [00:00<?, ?B/s]
model.safetensors: 0%|          | 0.00/90.9M [00:00<?, ?B/s]
tokenizer_config.json: 0%|          | 0.00/350 [00:00<?, ?B/s]
vocab.txt: 0.00B [00:00, ?B/s]
tokenizer.json: 0.00B [00:00, ?B/s]
special_tokens_map.json: 0%|          | 0.00/112 [00:00<?, ?B/s]
config.json: 0%|          | 0.00/190 [00:00<?, ?B/s]
```

```
In [76]: # Example sentences to embed
sentences = [
    "Natural language processing is a fascinating field of study.",
    "NLP combines computer science and linguistics to understand human langua",
    "Machine learning models can process and analyze text data effectively.",
    "The weather is sunny today.",
    "I love eating pizza for dinner.",
    "Deep learning has revolutionized artificial intelligence."
]
```

```
print("Input sentences:")
for i, sentence in enumerate(sentences):
    print(f"{i+1}. {sentence}")
```

Input sentences:

1. Natural language processing is a fascinating field of study.
2. NLP combines computer science and linguistics to understand human language.
3. Machine learning models can process and analyze text data effectively.
4. The weather is sunny today.
5. I love eating pizza for dinner.
6. Deep learning has revolutionized artificial intelligence.

```
In [77]: # Generate embeddings for all sentences
embeddings = model.encode(sentences)

print(f"\nEmbeddings shape: {embeddings.shape}")
print(f"Each sentence is represented as a {embeddings.shape[1]}-dimensional")
print(f"\nFirst sentence embedding (first 10 dimensions): {embeddings[0][:10]}
```

Embeddings shape: (6, 384)

Each sentence is represented as a 384-dimensional vector

First sentence embedding (first 10 dimensions): [ 0.03529549 -0.06365335 0.06940515 -0.00604334 -0.00801525 -0.01137893 0.01168134 -0.00718101 0.03682818 -0.00525783]

## Key Points about SentenceTransformers:

1. **Easy to Use:** With just a few lines of code, you can generate high-quality embeddings for any text.
2. **Pre-trained Models:** Many models are available for different use cases:
  - all-MiniLM-L6-v2 : Good balance of speed and performance
  - all-mpnet-base-v2 : Higher quality but slower
  - paraphrase-MiniLM-L6-v2 : Optimized for paraphrase detection
3. **Batch Processing:** The `encode()` method can handle single strings, lists of strings, or even large datasets efficiently.
4. **Consistent Output:** Unlike raw transformer models, SentenceTransformers are specifically fine-tuned to produce meaningful sentence-level representations.

## 5.3. Computing Semantic Similarity

One of the most powerful applications of text embeddings is computing semantic similarity between texts. We can use cosine similarity to measure how similar two embeddings are, which corresponds to how semantically similar the original texts are.

```
In [78]: from sklearn.metrics.pairwise import cosine_similarity
```



```
# Compute cosine similarity between all pairs of sentences
similarity_matrix = cosine_similarity(embeddings)

print("Cosine Similarity Matrix:")
print("Rows and columns correspond to sentences 1-6")
print(similarity_matrix.round(3))
```

```
Cosine Similarity Matrix:
Rows and columns correspond to sentences 1-6
[[ 1.      0.543  0.493 -0.031  0.139  0.367]
 [ 0.543  1.      0.493  0.043  0.079  0.277]
 [ 0.493  0.493  1.      0.005  0.024  0.354]
 [-0.031  0.043  0.005  1.      0.059 -0.104]
 [ 0.139  0.079  0.024  0.059  1.      0.041]
 [ 0.367  0.277  0.354 -0.104  0.041  1.     ]]
```

```
In [79]: # Let's find the most similar pairs (excluding self-similarity)
def find_most_similar_pairs(similarity_matrix, sentences, top_k=3):
    """Find the most similar sentence pairs"""
    n = len(sentences)
    similarities = []

    for i in range(n):
        for j in range(i+1, n): # Only consider upper triangle to avoid dup
            similarities.append((i, j, similarity_matrix[i][j]))

    # Sort by similarity score in descending order
    similarities.sort(key=lambda x: x[2], reverse=True)

    print(f"Top {top_k} most similar sentence pairs:")
    for i, (idx1, idx2, score) in enumerate(similarities[:top_k]):
        print(f"\n{i+1}. Similarity: {score:.3f}")
        print(f"    Sentence {idx1+1}: {sentences[idx1]}")
        print(f"    Sentence {idx2+1}: {sentences[idx2]}")

    find_most_similar_pairs(similarity_matrix, sentences)
```

Top 3 most similar sentence pairs:

1. Similarity: 0.543  
 Sentence 1: Natural language processing is a fascinating field of study.  
 Sentence 2: NLP combines computer science and linguistics to understand human language.
2. Similarity: 0.493  
 Sentence 2: NLP combines computer science and linguistics to understand human language.  
 Sentence 3: Machine learning models can process and analyze text data effectively.
3. Similarity: 0.493  
 Sentence 1: Natural language processing is a fascinating field of study.  
 Sentence 3: Machine learning models can process and analyze text data effectively.

```
In [80]: # Let's also compute similarity for a new query sentence
query = "Artificial intelligence and machine learning are transforming techn
```

```

query_embedding = model.encode([query])

# Compute similarity between query and all original sentences
query_similarities = cosine_similarity(query_embedding, embeddings)[0]

print(f"Query: {query}")
print("\nSimilarity to original sentences:")
for i, (sentence, similarity) in enumerate(zip(sentences, query_similarities)):
    print(f"{i+1}. {similarity:.3f} - {sentence}")

```

Query: Artificial intelligence and machine learning are transforming technology.

Similarity to original sentences:

1. 0.297 - Natural language processing is a fascinating field of study.
2. 0.381 - NLP combines computer science and linguistics to understand human language.
3. 0.422 - Machine learning models can process and analyze text data effectively.
4. -0.017 - The weather is sunny today.
5. 0.022 - I love eating pizza for dinner.
6. 0.618 - Deep learning has revolutionized artificial intelligence.

## Understanding Cosine Similarity

Cosine similarity measures the cosine of the angle between two vectors in the embedding space:

- **1.0:** Vectors point in exactly the same direction (identical meaning)
- **0.0:** Vectors are orthogonal (no semantic relationship)
- **-1.0:** Vectors point in opposite directions (opposite meaning)

In practice, most sentence embeddings produce similarity scores between 0.0 and 1.0, where:

- **> 0.7:** High semantic similarity
- **0.4 - 0.7:** Moderate semantic similarity
- **< 0.4:** Low semantic similarity

This makes cosine similarity particularly useful for:

- **Semantic Search:** Finding documents most similar to a query
- **Clustering:** Grouping similar texts together
- **Duplicate Detection:** Identifying near-duplicate content
- **Recommendation Systems:** Suggesting similar content

## Practical Applications of Text Embeddings

Text embeddings have numerous applications in NLP and beyond:

1. **Information Retrieval:** Build semantic search engines that understand meaning rather than just keyword matching.
2. **Text Classification:** Use embeddings as features for classifying documents, emails, or social media posts.
3. **Clustering and Topic Modeling:** Group similar documents together to discover themes or topics in large text collections.
4. **Paraphrase Detection:** Identify when two sentences express the same idea using different words.
5. **Question Answering:** Find the most relevant passages to answer a given question.
6. **Recommendation Systems:** Recommend articles, products, or content based on textual descriptions.
7. **Cross-lingual Tasks:** Many modern embedding models work across multiple languages, enabling multilingual applications.

```
In [81]: # Example: Simple semantic search function
def semantic_search(query, documents, model, top_k=3):
    """
    Perform semantic search to find the most relevant documents for a query.

    Args:
        query (str): The search query
        documents (list): List of documents to search through
        model: SentenceTransformer model
        top_k (int): Number of top results to return

    Returns:
        list: Top-k most relevant documents with their similarity scores
    """
    # Encode query and documents
    query_embedding = model.encode([query])
    doc_embeddings = model.encode(documents)

    # Compute similarities
    similarities = cosine_similarity(query_embedding, doc_embeddings)[0]

    # Get top-k results
    top_indices = np.argsort(similarities)[::-1][:top_k]

    results = []
    for idx in top_indices:
        results.append({
            'document': documents[idx],
            'similarity': similarities[idx],
            'index': idx
        })
```

```

    return results

# Example usage
documents = [
    "Python is a popular programming language for data science and machine l",
    "The stock market experienced significant volatility this week.",
    "Climate change is affecting weather patterns around the world.",
    "Deep learning models require large amounts of training data.",
    "The new restaurant downtown serves excellent Italian cuisine.",
    "Artificial neural networks are inspired by biological neural networks.",
    "Solar energy is becoming more cost-effective than fossil fuels."
]

search_query = "What programming languages are good for AI?"
results = semantic_search(search_query, documents, model)

print(f"Search Query: {search_query}")
print("\nTop Results:")
for i, result in enumerate(results):
    print(f"{i+1}. Score: {result['similarity']:.3f}")
    print(f"    Document: {result['document']}\n")

```

Search Query: What programming languages are good for AI?

Top Results:

1. Score: 0.491

Document: Python is a popular programming language for data science and machine learning.

2. Score: 0.366

Document: Artificial neural networks are inspired by biological neural networks.

3. Score: 0.214

Document: Deep learning models require large amounts of training data.

## Key Takeaways

1. **Text embeddings** transform discrete text into continuous vector representations that capture semantic meaning.
2. **SentenceTransformers** provides an easy-to-use interface for generating high-quality sentence embeddings with pre-trained models.
3. **Cosine similarity** is the standard metric for measuring semantic similarity between text embeddings.
4. **Practical applications** include semantic search, text classification, clustering, and recommendation systems.
5. **Model selection** matters - different SentenceTransformer models are optimized for different tasks and offer trade-offs between speed and quality.

Text embeddings are a fundamental building block for many modern NLP applications, and SentenceTransformers makes them accessible for both research and production use cases.

## Summary

In this tutorial, you were introduced to the HuggingFace ecosystem, a powerful set of tools and libraries designed to streamline the process of working with natural language processing (NLP) models. Here's a recap, including additional nuances that highlight the advantages of HuggingFace:

### 1. Datasets Library:

- **Efficient Data Handling:** The `datasets` library simplifies the process of loading, filtering, and transforming datasets, allowing you to manage large-scale data with ease.
- **Integration with PyTorch:** We learned how to integrate HuggingFace datasets with PyTorch `DataLoader`, ensuring seamless batching and processing of data, including the ability to customize batch processing with techniques like padding and truncation.

### 2. Transformers Library:

- **Model Utilization:** The `transformers` library provides access to a vast collection of pre-trained models, enabling you to easily apply state-of-the-art NLP models to your tasks.
- **Tokenization:** Tokenizers are essential for preparing text data for models. Each model has its tokenizer, tailored to its architecture, ensuring accurate text processing. We also learned how different tokenizers impact model behavior, highlighting the importance of using the correct tokenizer.
- **Sampling Techniques:** In text generation tasks, We explored different sampling techniques such as top-k and top-p (nucleus) sampling. These techniques allow for more creative and diverse text outputs by introducing controlled randomness, which is particularly useful in research scenarios that require nuanced text generation.

### 3. Evaluate Library:

- **Standardized Metrics:** The `evaluate` library offers a variety of standardized evaluation metrics, simplifying the process of assessing model performance, particularly in text generation tasks. This integration ensures that our evaluation process is consistent and aligned with academic standards.

### 4. Text Embeddings and SentenceTransformers:

- **Semantic Representations:** We explored how text embeddings transform discrete text into continuous vector representations that capture semantic meaning, enabling machines to understand and process human language more effectively.
- **SentenceTransformers Integration:** The `sentence-transformers` library provides an easy-to-use interface for generating high-quality sentence embeddings with pre-trained models optimized for semantic similarity tasks.
- **Practical Applications:** We demonstrated how to compute semantic similarity using cosine similarity, implement semantic search functionality, and apply embeddings to real-world tasks like document retrieval, clustering, and recommendation systems.
- **Model Selection:** We learned about different embedding models (e.g., `all-MiniLM-L6-v2`, `all-mpnet-base-v2`) and their trade-offs between speed and quality, helping you choose the right model for your specific use case.

#### 5. Ease of Training and Experimentation:

- **Flexible Training Loops:** HuggingFace offers both high-level abstractions through the Trainer API and the flexibility to implement custom training loops. This dual approach allows researchers to leverage easy-to-use training procedures while also providing the freedom to experiment with custom optimizations, such as different loss functions or gradient accumulation strategies.
- **Device Switching:** The seamless device switching capability in HuggingFace, where models and data can be easily moved between CPU, GPU (NVIDIA or Apple's MPS), and other accelerators, simplifies experimentation across different hardware setups. This flexibility is crucial in academic environments where computational resources may vary.

#### 6. Multilingual and Multimodal Support:

- **Multilingual Capabilities:** HuggingFace supports models that can process and generate text in multiple languages, which is crucial for academic research focused on cross-lingual NLP or low-resource languages.
- **Multimodal Research:** The availability of models that handle text, images, and audio simultaneously supports interdisciplinary research, enabling the exploration of how different data modalities interact and contribute to richer, more comprehensive AI models.

#### 7. Educational and Research Support:

- **Comprehensive Documentation and Tutorials:** HuggingFace's extensive documentation, tutorials, and example notebooks serve as valuable resources for both learning and teaching. These materials provide clear and accessible guidance on implementing complex NLP tasks, making it easier for students and researchers to grasp advanced concepts.

- **Structured Learning Pathways:** For academic institutions, HuggingFace offers structured learning pathways that can be incorporated into course curricula, helping students gain practical experience with state-of-the-art NLP techniques.

## Key Learnings

- **Efficiency and Abstraction:** HuggingFace libraries abstract many low-level details, allowing us to focus on higher-level model applications and research ideas without getting bogged down by implementation complexities.
- **Custom Training and Flexibility:** The ability to implement custom training loops or use the Trainer API offers flexibility in how you approach model training, allowing you to tailor your experiments precisely to your research needs.
- **Seamless Hardware Integration:** HuggingFace's support for easy device switching ensures that we can efficiently leverage available computational resources, whether on CPU, GPU, or other accelerators, without needing extensive code modifications.
- **Evaluation and Validation:** Consistent and standardized evaluation metrics are essential for assessing model performance and ensuring reproducibility in your research. HuggingFace's integration of these metrics simplifies the evaluation process and aligns with academic best practices.
- **Text Embeddings for Semantic Understanding:** Text embeddings provide a powerful way to capture semantic meaning in numerical form, enabling sophisticated applications like semantic search, clustering, and similarity comparison that go beyond simple keyword matching.
- **Support for Multilingual and Multimodal Research:** HuggingFace's support for multilingual and multimodal models makes it an invaluable tool for cross-lingual studies and interdisciplinary research that requires handling different types of data.

## Conclusion

HuggingFace provides a comprehensive and user-friendly platform for academic research in NLP, offering tools that simplify everything from data handling and model training to evaluation, deployment, and semantic understanding. Its flexibility, advanced features like custom sampling, ease of training loop implementation, seamless device integration, and powerful text embedding capabilities make it an indispensable resource for researchers looking to push the boundaries of natural language processing. The addition of SentenceTransformers extends the ecosystem's

capabilities to semantic similarity tasks, enabling researchers to build sophisticated applications that understand meaning rather than just process text superficially. As we continue to explore and utilize HuggingFace in your projects, these tools will not only enhance your productivity but also the quality and impact of our academic research.