CLASSIFICATION

```
text = """ code geass is one of those series that everybody recomends and once you watch it you know
 why.
Mechas, war, romance, comedy, rivality, unexpected twsits, explotions, mindgames and more mechas!
 what else do you want!!! xD it totally deserver its place as one of the best animes ever made.

the only aspect i see as a flaw is the art with a 9/10 the people seem a little bit deformed or
too skinny but once you get use to it is great!.The user wants freedom.

and it has by far the best story i have ever seen you just get hooked to it from the second
episode and cant stop until finishing it.

almost perfect serie, recomended to anyone who is a human beeing (nevermind, my dog liked it) """


!pip install transformers
!pip install datasets
!pip install sentencepiece
```

Transformers:

- The Transformers library, developed by Hugging Face, provides state-of-the-art natural language processing (NLP) models, including pre-trained models like BERT, GPT, RoBERTa, and more.

- These models are built using transformer architectures, which have revolutionized NLP by achieving superior performance on a wide range of tasks such as text classification, question answering, text generation, and language translation.

- The Transformers library allows users to easily load, fine-tune, and use these pre-trained models for various NLP tasks, enabling researchers and developers to leverage the latest advancements in deep learning for their projects.

- It also provides interfaces for working with these models in popular deep learning frameworks like TensorFlow and PyTorch.

Datasets:

- The Datasets library, also developed by Hugging Face, provides a collection of high-quality datasets for training and evaluating NLP models.

- These datasets cover a wide range of tasks and domains, including text classification, named entity recognition, machine translation, and more.

- The Datasets library offers a unified interface for accessing and manipulating these datasets, making it easy for researchers and practitioners to experiment with different datasets and benchmark their models against established baselines.

- It provides functionality for downloading datasets, splitting them into train/validation/test sets, preprocessing text data, and loading datasets in a format compatible with popular deep learning frameworks.

SentencePiece:

- SentencePiece is a library developed by Google for tokenization and text normalization tasks, particularly in the context of neural network-based models.

- It offers methods for segmenting text into subword units, which can be useful for handling out-of-vocabulary words, reducing vocabulary size, and improving model generalization.

- SentencePiece uses the unigram language model to learn a tokenization scheme from the input text data, allowing it to effectively handle various languages and writing systems.

-Training the SentencePiece Model:

- Imagine we have a big collection of text, like books or articles.
- We use this text to teach the SentencePiece model how words are formed.
- The model learns patterns in the text and decides how to split words into smaller parts called subwords.
- Tokenization of New Text:

  After training, we have a model that knows how to break down words into smaller pieces. When we give it a new sentence, like "The cat is sitting on the mat," it splits it into these smaller pieces based on what it learned during training. For example, it might split "sitting" into "sit" and "ting" because it didn't see "sitting" before, but it knows "sit" and "ting." Example Tokenization:

  Let's say our SentencePiece model knows the subwords "the," "cat," "is," "sit," "ting," "on," "mat," and ".". When it tokenizes "The cat is sitting on the mat," it might output: ["the", "cat", "is", "sit", "ting", "on", "the", "mat", "."]. Notice how it split "sitting" into "sit" and "ting" because it didn't know "sitting" as a whole. Handling New Words:

If the model encounters a word it didn't see during training, it tries to represent it using the subwords it knows. For example, if it sees a new word like "running," it might split it into "run" and "ning." In short, SentencePiece learns from a lot of text to understand how words are built and then uses that knowledge to break down new text into smaller pieces. This helps it handle new words and make text easier for computers to understand.

- This library is commonly used in conjunction with other NLP libraries and frameworks, such as TensorFlow and PyTorch, to preprocess text data before training or inference with deep learning models.

```
import datasets
import huggingface_hub
import matplotlib.pyplot as plt
import transformers
```

- huggingface_hub democratizes access to cutting-edge NLP models and datasets by providing a centralized repository where users can easily discover and download resources for their projects.
- By providing a centralized hub for hosting models and datasets, huggingface_hub streamlines the process of sharing and accessing resources, thereby promoting collaboration, reproducibility, and innovation in the field of NLP.

```
# Text Classification
from transformers import pipeline
classifier = pipeline('text-classification')
```

Transformers Pipeline

- The core architecture behind the text classification pipeline in the Hugging Face Transformers library typically involves a pre-trained transformer model fine-tuned on a large corpus of text data for the specific task of text classification. Here's how it works, illustrated with an example:

Loading the Model:

- When you create a text classification pipeline using pipeline('text-classification'), the Hugging Face library automatically downloads the pre-trained model associated with text classification tasks. This model is typically a transformer-based architecture like BERT, RoBERTa, or DistilBERT that has been fine-tuned on a dataset for text classification.

Tokenization:

- The input text provided to the pipeline is tokenized using the same tokenizer that was used during the pre-training of the model. This tokenizer converts the input text into a sequence of tokens that can be

processed by the transformer model.

Encoding:

- The tokenized input text is then encoded into numerical representations suitable for input to the transformer model. This encoding typically involves mapping each token to its corresponding index in the model's vocabulary and converting the input sequence into numerical vectors.

Model Inference:

- The encoded input sequence is fed into the pre-trained transformer model. The model processes the input sequence through multiple layers of self-attention mechanisms and feed-forward neural networks to extract contextual representations of the text.

Classification Head:

- After processing the input text through the transformer layers, a classification head is attached to the model. This classification head consists of one or more dense layers that take the final contextual representation of the input text as input and produce predictions for the classification task.

- For example, in binary classification tasks, the classification head may consist of a single neuron with a sigmoid activation function that outputs a probability score indicating the likelihood of the input text belonging to a particular class.

- In multi-class classification tasks, the classification head may consist of multiple neurons corresponding to each class, with a softmax activation function applied to produce a probability distribution over the classes. Prediction:

Finally, the model outputs the predicted class label or probability distribution over the classes for the input text. This prediction is returned as the output of the text classification pipeline.

config.json:

- This file contains the configuration parameters of the model, including details such as the model architecture, hyperparameters, tokenizer settings, and any other model-specific configurations. It provides essential information needed to instantiate and use the model in your code.
    - {
        - "attention_probs_dropout_prob": 0.1, This parameter determines the dropout probability for attention weights. Dropout is a regularization technique used during training to prevent overfitting by randomly setting some activations to zero.
        - "hidden_dropout_prob": 0.1, Similar to attention_probs_dropout_prob, this parameter specifies the dropout probability for the hidden layers. It controls the dropout applied to the outputs of each layer in the model.

- "hidden_size": 768, This parameter defines the dimensionality of the hidden layers in the transformer model. It represents the number of neurons in each layer's feedforward network.
- "num_attention_heads": 12,
- "num_hidden_layers": 12, The total number of layers in the transformer model, including both encoder and decoder layers. Each layer consists of sublayers like self-attention, feedforward, and layer normalization.
- "type_vocab_size": 2,
- "vocab_size": 30522, ... }

model.safetensors

- safetensors is a safe and fast file format for storing and loading tensors. Typically, PyTorch model weights are saved or pickled into a .bin file with Python's pickle utility. However, pickle is not secure and pickled files may contain malicious code that can be executed. safetensors is a secure alternative to pickle, making it ideal for sharing model weights.

Shopping cart data

shopping_cart = {

- "items": [{"name": "Laptop", "quantity": 1, "price": 999.99}, {"name": "Headphones", "quantity": 2, "price": 49.99}, {"name": "Mouse", "quantity": 1, "price": 19.99}], "total": 1119.96 }

import json

Serialize the shopping cart to JSON

- serialized_cart = json.dumps(shopping_cart)
- Save the serialized data to a file

  with open("shopping_cart.json", "w") as file: file.write(serialized_cart)

Read the serialized data from the file

with open("shopping_cart.json", "r") as file: serialized_cart = file.read()

Deserialize the JSON data into a Python dictionary

restored_cart = json.loads(serialized_cart)

print(restored_cart)

pytorch_model.bin:

- This file contains the actual weights and parameters of the BERT model, learned during pre-training or fine-tuning. Example content: Serialized tensor data representing the weights of each layer, attention

matrices, and other parameters of the model.

- Serialization: Serialization is the process of converting complex data structures, such as tensors, into a format that can be easily stored, transmitted, or reconstructed later. Serialization typically involves converting the data into a byte stream or a string format that preserves the structure and content of the original data.

import pickle

Example data to serialize

- data = {'name': 'John', 'age': 30, 'city': 'New York'}

Serialize data using pickle and save to a file

with open('data.pkl', 'wb') as file: pickle.dump(data, file)

Deserialize data from the file

with open('data.pkl', 'rb') as file: loaded_data = pickle.load(file)

print(loaded_data)

tokenizer_config.json:

- This file contains the configuration parameters of the BERT tokenizer, such as the tokenizer type, special tokens, vocabulary size, etc. Example content:

{

- "max_len": 512, This indicates the maximum sequence length that the tokenizer will process. Sequences longer than this length will be truncated, and sequences shorter than this length will be padded to reach this length.
- "model_type": "bert",
- "pad_token_id": 0, This is the token ID used for padding sequences. When sequences are shorter than the maximum length, they are padded with this token ID to match the maximum length. "vocab_size": 30522, ... }

```
classifier(text)
```

```
[{'label': 'POSITIVE', 'score': 0.9967235922813416}]
```

NER - Named Entity Recoginition

```
import pandas as pd

text1 = "Narendra Modi is prime minister of India"

ner = pipeline('ner', aggregation_strategy='simple')
out = ner(text1)
print(pd.DataFrame(out))
```

In the Hugging Face Transformers library, the aggregation_strategy parameter in the Named Entity Recognition (NER) pipeline specifies how multiple predictions for the same entity across different tokens are aggregated into a single entity span.

Let's illustrate the **aggregation_strategy='simple'** parameter in the Named Entity Recognition (NER) pipeline with an example:

Suppose we have the following input text:

```
"The company Google is headquartered in Mountain View, California."
```

And let's assume the NER model predicts the following entity labels for each token:

```
Token:         The   company  Google   is    headquartered   in    Mountain    View       ,        Californi
Predicted NER: O     O        B-ORG    O     O              O     B-LOC       I-LOC      O        B-LOC
```

The output of the pipeline would be a list of dictionaries, with each dictionary representing an entity found in the input text. For example:

```
[
  {'entity': 'ORG', 'text': 'Google', 'start': 12, 'end': 18},
  {'entity': 'LOC', 'text': 'Mountain View', 'start': 34, 'end': 47},
  {'entity': 'LOC', 'text': 'California', 'start': 53, 'end': 63}
]
```

Question Answering

```
reader = pipeline('question-answering')
question = 'what does user want ?'
outputs = reader(question=question,context = text)
pd.DataFrame([outputs])
```

Summarization

```
summarizer = pipeline('summarization')
outputs = summarizer( text, clean_up_tokenization_spaces=True, max_length=90)
print(outputs[0]['summary_text'])
```

clean_up_tokenization_spaces=True:

- When clean_up_tokenization_spaces is set to True, it indicates that the pipeline should remove any extra spaces resulting from tokenization before generating the summary text.
- Tokenization of text involves splitting it into individual tokens (words or subwords). Sometimes, tokenization can introduce extra spaces, especially around punctuation marks or special characters.
- Setting clean_up_tokenization_spaces=True ensures that these extra spaces are removed from the input text before it is summarized, resulting in cleaner and more readable summaries.

max_length=90:

- The max_length parameter specifies the maximum length (in tokens) of the generated summary text.
- When summarizing text, it's often desirable to limit the length of the summary to make it concise and focused.
- By setting max_length=90, you're instructing the summarization pipeline to generate a summary with a maximum length of 90 tokens. If the summary exceeds this length, it will be truncated to meet the specified limit.
- Limiting the length of the summary can help control the amount of information presented to the reader and ensure that the summary remains informative and easy to comprehend.

Translation

```
translate = pipeline('translation_en_to_de', model='Helsinki-NLP/opus-mt-en-de')
output = translate(text,clean_up_tokenization_spaces=True,min_length=120)
print(output[0]['translation_text'])
```

TEXT CLASSIFICATION WITH HUGGINGFACE

```
!pip install --upgrade pip
!pip install transformers
!pip install datasets
!pip install sentencepiece
```

Datasets

```
# Emotion detector
from datasets import list_datasets
all_datasets = list_datasets()
print(f"There are total of {len(all_datasets)} in the hub")
print(f"Some example datasets are {all_datasets[:5]}")
```

```
from datasets import load_dataset
emotion_dataset = load_dataset('emotion')
emotion_dataset
```

```
train_dataset = emotion_dataset['train']
train_dataset
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 16000
})
```

```
train_dataset[0]
```

```
{'text': 'i didnt feel humiliated', 'label': 0}
```

```
train_dataset[:5]
```

```
{'text': ['i didnt feel humiliated',
  'i can go from feeling so hopeless to so damned hopeful just from being around someone who
cares and is awake',
  'im grabbing a minute to post i feel greedy wrong',
  'i am ever feeling nostalgic about the fireplace i will know that it is still on the
property',
  'i am feeling grouchy'],
 'label': [0, 0, 3, 2, 3]}
```

```
train_dataset.column_names
```

```
['text', 'label']
```

```
train_dataset.features
```

```
{'text': Value(dtype='string', id=None),
 'label': ClassLabel(names=['sadness', 'joy', 'love', 'anger', 'fear', 'surprise'], id=None)}
```

```
train_dataset.features['label'].names
```

```
['sadness', 'joy', 'love', 'anger', 'fear', 'surprise']
```

!wget https://raw.githubusercontent.com/nachikethmurthy/Source-Code-Dataset-for-Machine-Learning-using-Python/main/Data/sentiment_train

```
!wget https://raw.githubusercontent.com/nachikethmurthy/Source-Code-Dataset-for-Machine-Learning-usi
```

```
--2024-04-22 03:56:17--  https://raw.githubusercontent.com/nachikethmurthy/Source-Code-Dataset-
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... conn
HTTP request sent, awaiting response... 200 OK
Length: 447555 (437K) [text/plain]
Saving to: 'sentiment_train'

sentiment_train     100%[===================>] 437.07K  --.-KB/s    in 0.03s

2024-04-22 03:56:18 (15.7 MB/s) - 'sentiment_train' saved [447555/447555]
```

```
# load the dataset
sentiment_train = load_dataset('csv',data_files = "sentiment_train",sep='\t')
sentiment_train
```

```
Generating train split:        6918/0 [00:00<00:00, 178308.82 examples/s]
```

```
DatasetDict({
    train: Dataset({
        features: ['sentiment', 'text'],
        num_rows: 6918
    })
})
```

```
emotion_dataset
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 16000
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
```

```
    })
})
```

Convert the Hugging Face Dataset object (emotion_dataset) into a pandas DataFrame (df)

```
import pandas as pd
emotion_dataset.set_format(type='pandas')
df = emotion_dataset['train'][:]
df.head()
```

	text	label
0	i didnt feel humiliated	0
1	i can go from feeling so hopeless to so damned...	0
2	im grabbing a minute to post i feel greedy wrong	3
3	i am ever feeling nostalgic about the fireplac...	2
4	i am feeling grouchy	3

Next steps: Generate code with df View recommended plots

```
emotion_dataset['train']
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 16000
})
```

```
emotion_dataset['train'][:]
```

	text	label
0	i didnt feel humiliated	0
1	i can go from feeling so hopeless to so damned...	0
2	im grabbing a minute to post i feel greedy wrong	3
3	i am ever feeling nostalgic about the fireplac...	2
4	i am feeling grouchy	3
...	...	...
15995	i just had a very brief time in the beanbag an...	0
15996	i am now turning and i feel pathetic that i am...	0
15997	i feel strong and good overall	1
15998	i feel like this was such a rude comment and i...	3
15999	i know a lot but i feel so stupid because i ca...	0

16000 rows × 2 columns

```
emotion_dataset['train'].features['label'].int2str(0)
```

```
'sadness'
```

The int2str()

method is used to convert integer indices back to string labels. It's particularly useful when dealing with classification tasks where labels are represented as integers but you want to convert them back to their original string representations.

```
emotion_dataset['train'].features['label'].int2str(1)
```

```
'joy'
```

```
def label_to_str(row):
  return emotion_dataset['train'].features['label'].int2str(row)

df['labels_name'] = df['label'].apply(label_to_str)
df.head()
```

	text	label	labels_name
0	i didnt feel humiliated	0	sadness
1	i can go from feeling so hopeless to so damned...	0	sadness
2	im grabbing a minute to post i feel greedy wrong	3	anger
3	i am ever feeling nostalgic about the fireplac...	2	love
4	i am feeling grouchy	3	anger

Next steps: Generate code with df View recommended plots

```
emotion_dataset.reset_format()   #Hugging Face datasets library, such as the DatasetDict
```

Tokenization - Character tokenization

Tokenization - Word tokenization

Goal --> Find a way to effienctly perform the tokenization - Subword Tokenization

```
from transformers import AutoTokenizer
model = 'distilbert-base-uncased'
tokenizer = AutoTokenizer.from_pretrained(model)
```

```
tokenizer_config.json: 100%  28.0/28.0 [00:00<00:00, 1.72kB/s]
config.json: 100%  483/483 [00:00<00:00, 32.2kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 17.9MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 29.1MB/s]
```

```
text = "My favorite food is Masala Dosa"
encoded_text = tokenizer(text)
print(encoded_text)
```

```
{'input_ids': [101, 2026, 5440, 2833, 2003, 16137, 7911, 9998, 2050, 102], 'attention_mask': [1,
```

- input_ids: This is a list of token IDs representing the input text after tokenization. Each token in the input text is mapped to a corresponding ID from the model's vocabulary. Here's what each ID represents:

- 101: This is the special token [CLS], which stands for "classification" and is added at the beginning of the input sequence.
- 2026: The token ID for the word "My".
- 5440: The token ID for the word "favorite".
- 2833: The token ID for the word "food".
- 2003: The token ID for the word "is".
- 16137: The token ID for the subword "mas".
- 7911: The token ID for the subword "##ala".
- 9998: The token ID for the subword "dos".
- 2050: The token ID for the subword '##a'
- 102: This is another instance of the special token [SEP], indicating the end of the sequence.
- attention_mask: This is a list indicating which tokens in the input sequence should be attended to (have their attention scores calculated) and which ones should be ignored. In this case, all tokens have an attention mask value of 1, indicating that they should all be attended to.

[CLS] stands for "classification"

- The special token [CLS] stands for "classification" and is added at the beginning of the input sequence specifically for tasks that involve classification or sequence labeling, such as sentiment analysis, text classification, or named entity recognition. Its main purpose is to provide a representation of the entire input sequence that can be used for classification tasks.
- When training a model for classification tasks using techniques like BERT, the model typically receives a single vector representation of the entire input sequence. The [CLS] token's position is chosen because it always appears in the same position in every input sequence, making it a consistent representation to use for classification purposes

[SEP] Token (ID 102):

- The [SEP] token is indeed a separator token used to mark the end of a sequence or to separate segments within a sequence.
- It's added at the end of each input sequence to indicate the end of the sequence.
- In tasks like sentence classification or question answering, where there's only one input sequence, a single [SEP] token is added at the end to denote the end of that sequence.

```
type(encoded_text)
```

```
transformers.tokenization_utils_base.BatchEncoding
def __init__(data: Optional[Dict[str, Any]]=None, encoding: Optional[Union[EncodingFast,
Sequence[EncodingFast]]]=None, tensor_type: Union[None, str, TensorType]=None,
prepend_batch_axis: bool=False, n_sequences: Optional[int]=None)
```

```
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py
Holds the output of the [`~tokenization_utils_base.PreTrainedTokenizerBase.__call__`],
[`~tokenization_utils_base.PreTrainedTokenizerBase.encode_plus`] and
[`~tokenization_utils_base.PreTrainedTokenizerBase.batch_encode_plus`] methods (tokens, a
```

transformers:

This is the namespace or package where the class tokenization_utils_base is defined. In the context of Hugging Face's Transformers library, this is where the tokenization-related utilities are implemented.

tokenization_utils_base:

This is a module within the Transformers library that contains base classes and utilities for tokenization. It provides the foundation for various tokenization-related functionalities.

BatchEncoding:

This is the specific class or type of object returned by the tokenizer when encoding a batch of input sequences. It represents a batch of encoded sequences and typically contains attributes such as input_ids, attention_mask, and other relevant information about the encoded batch.

Double-click (or enter) to edit

```
tokens = tokenizer.convert_ids_to_tokens(encoded_text.input_ids)
print(tokens)
```

```
['[CLS]', 'my', 'favorite', 'food', 'is', 'mas', '##ala', 'dos', '##a', '[SEP]']
```

```
tokenizer.convert_tokens_to_string(tokens)
```

```
'[CLS] my favorite food is masala dosa [SEP]'
```

```
tokenizer.vocab_size
```

```
30522
```

```
tokenizer.model_max_length
```

```
512
```

```
tokenizer.model_input_names
```

```
['input_ids', 'attention_mask']
```

TOKENIZATION ON DATASET

```
emotion_dataset['train'][:2]
```

```
{'text': ['i didnt feel humiliated',
  'i can go from feeling so hopeless to so damned hopeful just from being around someone who
cares and is awake'],
 'label': [0, 0]}
```

```
def tokenization(batch):
  return tokenizer(batch['text'], padding=True, truncation=True)

tokenization(emotion_dataset['train'][:2])
```

```
{'input_ids': [[101, 1045, 2134, 2102, 2514, 26608, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0], [101, 1045, 2064, 2175, 2013, 3110, 2061, 20625, 2000, 2061, 9636, 17772, 2074, 2013,
2108, 2105, 2619, 2040, 14977, 1998, 2003, 8300, 102]], 'attention_mask': [[1, 1, 1, 1, 1, 1,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]}
```

- {'input_ids': [[101, 1045, 2134, 2102, 2514, 26608, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
- [101, 1045, 2064, 2175, 2013, 3110, 2061, 20625, 2000, 2061, 9636, 17772, 2074, 2013, 2108, 2105, 2619, 2040, 14977, 1998, 2003, 8300, 102]],
- 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
- [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]}

```
emotions_encoded = emotion_dataset.map(tokenization, batched=True, batch_size=None)
emotions_encoded
```

HuggingFace Models

BitsAndBytes:

The bitsandbytes library is a Python library developed by OpenAI for efficient data serialization and deserialization.

Here's how it works:

- Original List: [10, 20, 30, 40, 50]

- Serialized Form: 10 20 30 40 50

- In the serialized form, each number is represented as a string of characters separated by spaces. This representation may look shorter compared to the original list, but in terms of memory or storage size, it may not necessarily be shorter.

- The primary benefit of serialization with bitsandbytes is not to make the data physically shorter but to convert it into a format that is easier to handle, transmit, or store, especially when dealing with complex data structures or when interoperating with other systems.

Accelerate:

- Suppose you're training a deep learning model to recognize images. This process involves lots of complex calculations.

- With accelerate, you can harness the power of specialized hardware like GPUs (Graphics Processing Units) to speed up these calculations.

- For example, if it takes 10 hours to train your model on a regular computer, accelerate might reduce it to just 2 hours on a GPU.

- Similarly, when you're running your trained model to make predictions, accelerate helps make those predictions faster, which is crucial for real-time applications like video processing or natural language understanding.

  In summary, bitsandbytes helps with data compression and decompression, while accelerate boosts the performance of deep learning tasks by leveraging specialized hardware.

```
!pip install accelerate
```

Mixtral-8x22B-v0.1

- "8x22" indicates that the model consists of 8 layers of 22 blocks each.
- "B" typically represents billion parameters, suggesting that the model has a large number of parameters.
- Now, let's discuss the concept of a Mixture of Experts (MoE) and how it's trained:
- Mixture of Experts (MoE):

- A Mixture of Experts is a neural network architecture that consists of multiple subnetworks called "experts" working in parallel, along with a gating mechanism that learns to dynamically select which expert to use for a given input. Each expert specializes in different aspects of the data, and the gating mechanism determines the relevance or importance of each expert for a particular input.
- In the context of language modeling, each expert might specialize in generating specific types of text, such as formal language, informal language, technical jargon, etc.
- Training Process:
- Expert Training: Initially, each expert in the MoE is trained independently on the data. This allows each expert to specialize in different aspects of the data.
- Gating Network Training: Simultaneously, the gating network is trained to learn how to dynamically combine the outputs of the experts based on the input data. The gating network's goal is to select the most relevant expert for each input.
- Joint Training: After training the experts and the gating network separately, the entire MoE model is fine-tuned jointly. During this joint training, the parameters of both the experts and the gating network are updated together to optimize the overall performance of the model.
- Example:
- Consider a language generation task where the input is a prompt, and the model is tasked with generating a coherent continuation of the prompt.
- Expert Networks: The MoE might consist of experts trained on different genres of text, such as news articles, fiction, scientific papers, etc.
- Gating Mechanism: The gating network learns to identify the genre or style of the input prompt and dynamically selects the expert best suited to generate text in that style.
- Training: During training, the MoE is fed a variety of prompts from different genres, and the parameters of both the experts and the gating network are adjusted to minimize the generation loss across all inputs.
- Overall, the Mixture of Experts architecture enables the model to leverage the strengths of multiple specialized subnetworks, resulting in improved performance and versatility across different types of data.

Mixture of Experts Model Working with example

- Imagine you're planning a big event, like a party, and you need to organize various activities for different groups of guests. You decide to hire a team of experts, each specializing in a different aspect of event planning:
- Food Expert: This expert knows all about cooking and catering delicious meals.

- Music Expert: This expert specializes in creating playlists and setting up sound systems for entertainment.
- Decoration Expert: This expert is skilled at designing and decorating the venue to create a festive atmosphere.
- Entertainment Expert: This expert arranges games, performances, and other activities to keep guests entertained.
- Now, here's how the MoE architecture works:
- Expert Training: Each expert receives specialized training in their respective area. The food expert learns recipes, the music expert curates playlists, the decoration expert hones their design skills, and the entertainment expert plans fun activities.
- Gating Mechanism: You, as the event organizer, act as the gating mechanism. Based on the preferences and interests of your guests, you dynamically select which expert to consult for each aspect of the event. For example, if you have guests who love dancing, you might prioritize the music expert and choose lively playlists.
- Joint Training: As you plan the event, you interact with the experts and fine-tune the details. You might adjust the menu based on dietary preferences, tweak the playlist to suit different moods, and personalize the decorations to match the theme. This collaborative process optimizes the overall event experience.
    - In summary, the MoE model architecture combines the expertise of specialized subnetworks (the experts) with a dynamic selection mechanism (the gating mechanism) to optimize performance across different tasks or domains. By leveraging the strengths of multiple experts, the MoE can adapt to diverse inputs and produce high-quality outputs tailored to specific needs.

Take too much space more than 200gb (estimated)

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model_id = "mistral-community/Mixtral-8x22B-v0.1"
tokenizer = AutoTokenizer.from_pretrained(model_id)


model = AutoModelForCausalLM.from_pretrained(model_id)

text = "Hello my name is"
inputs = tokenizer(text, return_tensors="pt")

outputs = model.generate(**inputs, max_new_tokens=20)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

AutoModelForCausalLM and AutoTokenizer

These are classes provided by the Hugging Face transformers library. They are used to simplify the process of loading and using pre-trained language models and tokenizers.

AutoModelForCausalLM:

- This class is used to load pre-trained language models that are specifically designed for causal language modeling tasks.
- Causal language modeling is a type of language modeling where the model predicts the next token in a sequence given the preceding tokens. It is commonly used for tasks like text generation and completion.
- By using AutoModelForCausalLM, you can easily load a pre-trained model suitable for causal language modeling without worrying about the specific model architecture or configuration.

AutoTokenizer:

- This class is used to load pre-trained tokenizers that are compatible with specific language models.
- Tokenizers are used to preprocess text data into tokens that can be fed into a language model.
- The AutoTokenizer class automatically selects the appropriate tokenizer based on the specified model_id.

1. **Table Question Answering from Tasks**

```
from transformers import pipeline
import pandas as pd
```

```
# prepare table + question
data = {"Actors": ["Brad Pitt", "Leonardo Di Caprio", "George Clooney"], "Number of movies":
 ["87", "53", "69"]}
table = pd.DataFrame.from_dict(data)
question = "how many movies does Leonardo Di Caprio have?"
```

```
# pipeline model
# Note: you must to install torch-scatter first.
tqa = pipeline(task="table-question-answering", model="google/tapas-large-finetuned-wtq")
```

```
# result

print(tqa(table=table, query=question)['cells'][0])
#53
```

```
53
```

```
print(tqa(table=table, query="bradd pitt movies")['cells'][0])
```

```
87
```

2. **Document-Question-Answering from Tasks**

```
from transformers import pipeline
from PIL import Image

pipe = pipeline("document-question-answering", model="naver-clova-ix/donut-base-finetuned-docvqa")

question = "What is the purchase amount?"
image = Image.open("/content/Bollywood.jpg")

pipe(image=image, question=question)
```

```
[{'answer': 'inc42'}]
```

Evaluation Metrics

Bert Score

BERTScore is a metric used for evaluating the quality of text generated by language models. It measures how similar the generated text (predictions) is to the reference text.

Here's a simpler breakdown:

What it measures:

- BERTScore calculates a score for each token in the generated text (predictions) by comparing it to each token in the reference text. It uses pre-trained language models like BERT to understand the context of the words.

How it works:

- BERTScore looks at each word in the generated text and finds the most similar word in the reference text using cosine similarity. It then calculates precision, recall, and F1 score based on these similarities.

What it tells us:

- The precision score tells us how many words in the generated text match words in the reference text. The recall score tells us how many words in the reference text are captured by the generated text. The F1 score is a combination of precision and recall, providing an overall measure of similarity.

Usage:

- BERTScore can be used to compare different language generation models, evaluate the quality of machine translation, text summarization, and other natural language processing tasks.

In summary, BERTScore is a metric that measures the similarity between generated text and reference text, providing insight into the quality of language generation models. It's a useful tool for evaluating and comparing different text generation techniques.

```
# Install the bert_score library
!pip install bert_score

# Import the necessary functions
from bert_score import score

# Prepare your predictions and references
predictions = ["hello world", "general kenobi"]
references = ["hello world", "general kenobi"]

# Compute BERTScore specifying the language (English)
precision, recall, f1 = score(predictions, references, lang="en")

# Print the results
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
!pip install evaluate
```

```
from huggingface_hub import notebook_login

notebook_login()
```

Copy a token from your Hugging Face tokens page and paste it below.

Immediately click login after copying your token or it might be stored in plain text in this notebook file.

Token:

Add token as git credential?

Login

Pro Tip: If you don't already have one, you can create a dedicated 'notebooks' token with 'write' access, that you can then easily reuse for all notebooks.

```
from evaluate import load
bertscore = load("bertscore")
predictions = ["hello there", "general kenobi"]
references = ["hello there", "general kenobi"]
results = bertscore.compute(predictions=predictions, references=references, lang="en")
```

```
Downloading builder script: 100%  7.95k/7.95k [00:00<00:00, 131kB/s]
Some weights of RobertaModel were not initialized from the model checkpoint at roberta-large and
You should probably TRAIN this model on a down-stream task to be able to use it for predictions
```

BLEU (Bilingual Evaluation Understudy):

What it measures:

- BLEU is a metric used to evaluate the quality of machine-translated text by comparing it to human translations. It measures how closely the machine-generated text matches the human reference translations.

How it works:

- Segmentation:
  - BLEU first divides the candidate translation (machine-generated text) and reference translations (human-generated text) into segments, usually sentences.
- N-gram matching:
  - It then compares each segment of the candidate translation to the corresponding segments in the reference translations. BLEU considers the overlap of n-grams (contiguous sequences of n words) between the candidate and reference translations.
- Precision calculation:
  - For each n-gram size, BLEU calculates precision, which measures the proportion of n-grams in the candidate translation that appear in the reference translations.
- Brevity penalty:
  - BLEU also penalizes translations that are shorter than the reference translations to prevent shorter translations from receiving artificially high scores.
- BLEU score:
  - Finally, BLEU combines the precision scores for different n-gram sizes using a geometric mean and applies the brevity penalty to compute the overall BLEU score

Scoring range:

- BLEU scores range from 0 to 1, where a score closer to 1 indicates a better match between the machine-generated and human-translated texts. A score of 1 means the machine translation is identical to a human translation.

Usage:

- BLEU is commonly used for evaluating machine translation systems. It helps researchers and developers assess the quality of their translation models and make improvements.

In summary, BLEU provides a quantitative measure of the quality of machine translations by comparing them to human translations. Higher BLEU scores indicate better quality translations. It's a widely-used and cost-effective metric for assessing translation performance.

```
import nltk
from nltk.translate.bleu_score import sentence_bleu

# Define candidate and reference sentences
candidate = "hello world"
reference = ["hello world", "general kenobi"]

# Tokenize candidate and reference sentences
candidate_tokens = candidate.split()
reference_tokens = [ref.split() for ref in reference]

# Calculate BLEU score
bleu_score = sentence_bleu(reference_tokens, candidate_tokens)

# Print BLEU score
print("BLEU Score:", bleu_score)
```

```
BLEU Score: 1.491668146240062e-154
/usr/local/lib/python3.10/dist-packages/nltk/translate/bleu_score.py:552: UserWarning:
The hypothesis contains 0 counts of 3-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
  warnings.warn(_msg)
/usr/local/lib/python3.10/dist-packages/nltk/translate/bleu_score.py:552: UserWarning:
The hypothesis contains 0 counts of 4-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
  warnings.warn(_msg)
```

GLUE (General Language Understanding Evaluation)

- GLUE, or General Language Understanding Evaluation, is a test used to measure how well a machine understands human language. It includes different datasets to evaluate various aspects of language understanding.

To use GLUE:

Load the relevant metric:

- Depending on which part of GLUE you're testing against, you load the corresponding metric.

Calculate the metric:

- Once loaded, provide your model's predictions and the reference answers. The metric then computes

For example, if you're testing on a subset called MRPC, which checks accuracy and F1 score:

```
from evaluate import load

#Load the GLUE metric for the MRPC subset

glue_metric = load('glue', 'mrpc')

# Prepare your predictions and references

predictions = [0, 1] references = [0, 1]

# Compute the GLUE score

results = glue_metric.compute(predictions=predictions, references=references)

# Print the results

print(results)
```

For STSB, which checks Pearson and Spearman correlations:

```
from evaluate import load

# Load the GLUE metric for the STSB subset

glue_metric = load('glue', 'stsb')

# Prepare your predictions and references

predictions = [-10., -11., -12., -13., -14., -15.] references = [0., 1., 2., 3., 4., 5.]

# Compute the GLUE score

results = glue_metric.compute(predictions=predictions, references=references)

# Print the results
```