Tutorials on Stance Detection using Pre-trained Language Models: Fine-tuning BERT and Prompting Large Language Models

Yun-Shiuan Chuang[1,2]
[1]Department of Psychology, [2]Department of Computer Science
yunshiuan.chuang@wisc.edu

Abstract

This paper presents two self-contained tutorials on stance detection in Twitter data using BERT fine-tuning and prompting large language models (LLMs). The first tutorial explains BERT architecture and tokenization, guiding users through training, tuning, and evaluating standard and domain-specific BERT models with HuggingFace transformers. The second focuses on constructing prompts and few-shot examples to elicit stances from ChatGPT and open-source FLAN-T5 without fine-tuning. Various prompting strategies are implemented and evaluated using confusion matrices and macro F1 scores. The tutorials provide code, visualizations, and insights revealing the strengths of few-shot ChatGPT and FLAN-T5 which outperform fine-tuned BERTs. By covering both model fine-tuning and prompting-based techniques in an accessible, hands-on manner, these tutorials enable learners to gain applied experience with cutting-edge methods for stance detection.

1 Part 1: Stance Detection on Tweets with fine-tuning BERT

Note: This tutorial consists of two separate Python notebooks. This notebook is the first one. The second notebook can be found here. I recommend that you go through the first notebook before the second one as the second notebook builds on top of the first one.

1. First notebook (this one): Fine-tuning BERT models: include standard BERT and domain-specific BERT

- link: https://colab.research.google.com/drive/1nxziaKStwRnSyOLI6pLNBaAnB_aB6IsE?usp=sharing

2. Second notebook: Prompting large language models (LLMs): include ChatGPT, FLAN-T5 and different prompt types (zero-shot, few-shot, chain-of-thought)

- link: https://colab.research.google.com/drive/1IFr6Iz1YH9XBWUKcWZyTU-1QtxgYqrmX?usp=sharing

1.1 Getting Started: Overview, Prerequisites, and Setup

Objective of the tutorial: This tutorial will guide you through the process of stance detection on tweets using two main approaches: fine-tuning a BERT model and using large language models (LLMs).

Prerequisites:

- If you want to run the tutorial without editting the codes but want to understand the content
  - Basic Python skills: functions, classes, pandas, etc.
  - Basic ML knowledge: train-validation-test split, F1 score, forward pass, backpropagation etc.
- Familiarity with NLP concepts is a plus, particularly with transformers. However, if you're not familiar with them, don't worry. I'll provide brief explanations in the tutorial, as well as links to fantastic in-depth resources throughout the text.

Acknowledgements

- While the application of BERT on stance detection is my own work, some part of this tutorials, e.g., transformer and BERT, are inspired by the following tutorials. Some of the figures are also modified from the images in these tutorials. I highly recommend you to check them out if you want to learn more about transformers and BERT.
  - http://jalammar.github.io/illustrated-transformer/
  - http://jalammar.github.io/illustrated-bert/
- This tutorial was created with the assistance of ChatGPT (GPT-4), a cutting-edge language model developed by OpenAI. The AI-aided writing process involved an iterative approach, where I provided the model with ideas for each section and GPT-4 transformed those ideas into well-structured paragraphs. Even the outline itself underwent a similar iterative process to refine and improve the tutorial structure. Following this, I fact-checked and revised the generated content, asking GPT-4 to make further revisions based on my evaluation, until I took over and finalized the content.

Setup

1. Before we begin with Google Colab, please ensure that you have selected the GPU runtime. To do this, go to `Runtime` -> `Change runtime type` -> `Hardware accelerator` -> `GPU`. This will ensure that the note will run more efficiently and quickly.
2. Now, let's download the content of this tutorial and install the necessary libraries by running the following cell.

[1]:
```
from os.path import join
ON_COLAB = True
if ON_COLAB:
  !git clone --single-branch --branch colab https://github.com/yunshiuan/
 ↪prelim_stance_detection.git
  !python -m pip install pandas datasets openai accelerate transformers␣
 ↪transformers[sentencepiece] torch==1.12.1+cu113 -f https://download.pytorch.
 ↪org/whl/torch_stable.html emoji -q
  %cd /content/prelim_stance_detection/scripts
else:
  # if you are not on colab, you have to set up the environment by yourself. You␣
 ↪would also need a machine with GPU.
  %cd scripts
```

```
Cloning into 'prelim_stance_detection'...
remote: Enumerating objects: 513, done.
remote: Counting objects: 100% (36/36), done.
```

```
remote: Compressing objects: 100% (24/24), done.
remote: Total 513 (delta 21), reused 24 (delta 12), pack-reused 477
Receiving objects: 100% (513/513), 58.56 MiB | 12.29 MiB/s, done.
Resolving deltas: 100% (254/254), done.
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪492.4/492.4
kB 2.7 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪73.6/73.6 kB
2.4 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪244.2/244.2 kB
14.6 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪7.4/7.4 MB
43.5 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪1.8/1.8 GB
472.8 kB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪361.8/361.8 kB
28.6 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪115.3/115.3 kB
10.2 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪212.5/212.5 kB
13.5 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪134.8/134.8
kB 2.5 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------------
 ↪268.8/268.8 kB
```

```
18.5 MB/s eta 0:00:00
    ␣
 ↪---------------------------------------------------------------------------------------------
 ↪7.8/7.8 MB
48.2 MB/s eta 0:00:00
    ␣
 ↪---------------------------------------------------------------------------------------------
 ↪1.3/1.3 MB
59.0 MB/s eta 0:00:00
    ␣
 ↪---------------------------------------------------------------------------------------------
 ↪1.3/1.3 MB
63.9 MB/s eta 0:00:00
  Building wheel for emoji (pyproject.toml) ... done
ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the following
dependency conflicts.
torchaudio 2.0.2+cu118 requires torch==2.0.1, but you have torch 1.12.1+cu113
which is incompatible.
torchdata 0.6.1 requires torch==2.0.1, but you have torch 1.12.1+cu113 which is
incompatible.
torchtext 0.15.2 requires torch==2.0.1, but you have torch 1.12.1+cu113 which is
incompatible.
torchvision 0.15.2+cu118 requires torch==2.0.1, but you have torch 1.12.1+cu113
which is incompatible.
/content/prelim_stance_detection/scripts
```

[ ]:
```
# a helper function to load images in the notebook
from IPython.display import display
from PIL import Image as PILImage
from parameters_meta import ParametersMeta as par
PATH_IMAGES = join(par.PATH_ROOT, "images")


def display_resized_image_in_notebook(file_image, scale=1,␣
 ↪use_default_path=True):
    """ Display an image in a notebook.
    """
    # - https://stackoverflow.com/questions/69654877/
 ↪how-to-set-image-size-to-display-in-ipython-display
    if use_default_path:
```

```
        file_image = join(PATH_IMAGES, file_image)
    image = PILImage.open(file_image)
    display(image.resize((int(image.width * scale), int(image.height * scale))))
```

1.2 What is Stance Detection and Why is it Important?

Stance detection is an essential task in natural language processing that aims to determine the attitude expressed by an author towards a specific target, such as an entity, topic, or claim. The output of stance detection is typically a categorical label, such as "in-favor," "against," or "neutral," indicating the stance of the author in relation to the target. This task is critical for studying human belief dynamics, e.g., how people influence each other's opinions and how beliefs change over time. To better understand the complexities involved in stance detection, let's consider an example related to the topic of "abortion legalization".

For example, consider the following tweet:

"***A pregnancy, planned or unplanned, brings spouses, families & everyone closer to each other. #Life is beautiful! #USA***"

In this case, the stance expressed towards the topic of abortion legalization might be inferred as against, but the clues indicating the author's attitude are implicit and subtle - notice that it does not explicitly mention abortion, making it challenging to determine the stance without careful examination and contextual understanding.

There are two key challenges in stance detection, especially when working with large datasets like Twitter data. First, as illustrated above, the underlying attitude expressed in the text is often subtle, which requires domain knowledge and context to correctly label the stance. Second, the corpus can be very large, with millions of tweets, making it impractical to manually annotate all of them.

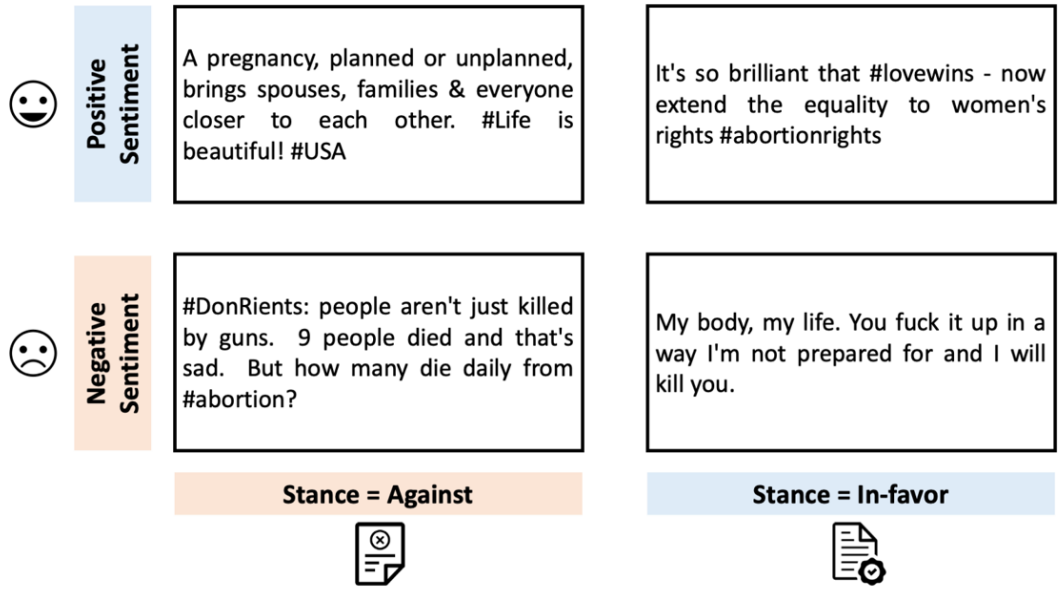
To address these challenges, we will leverage advanced natural language processing (NLP) techniques including two paradigms, 1) fine-tuning BERT model, and 2) prompting large language models (LLMs). I will elaborte the details of these two approaches in the following sections.

Before discussing the two paradigms for addressing the challenges in stance detection, it's essential to understand the difference between **sentiment analysis** and **stance detection**, as these two tasks are often confused.

Sentiment analysis involves identifying the overall emotional tone expressed in a piece of text, usually categorized as positive, negative, or neutral. In contrast, stance detection aims to determine the specific attitude of an author towards a target. While sentiment analysis focuses on the general emotional valence of the text, stance detection requires a deeper understanding of the author's position concerning the target topic.

To illustrate that sentiment and stance are orthogonal concepts, consider the following four examples, each representing a combination of two stance types (against and in-favor) and two sentiments (positive and negative):

[ ]:
```
display_resized_image_in_notebook("stance_vs_sentiment.png",0.7)
```



In this tutorial, we will focus on stance detection in the context of the "Abortion" topic using the SemEval-2016 dataset (data is publicly available here). We chose the abortion topic because it is currently a hotly debated issue, and it is important to understand public opinion on this matter. We will analyze a dataset containing tweets about abortion, with each tweet labeled as either in-favor, against, or neutral with respect to the topic. My goal is to develop a model that can accurately identify the stance expressed in these tweets.

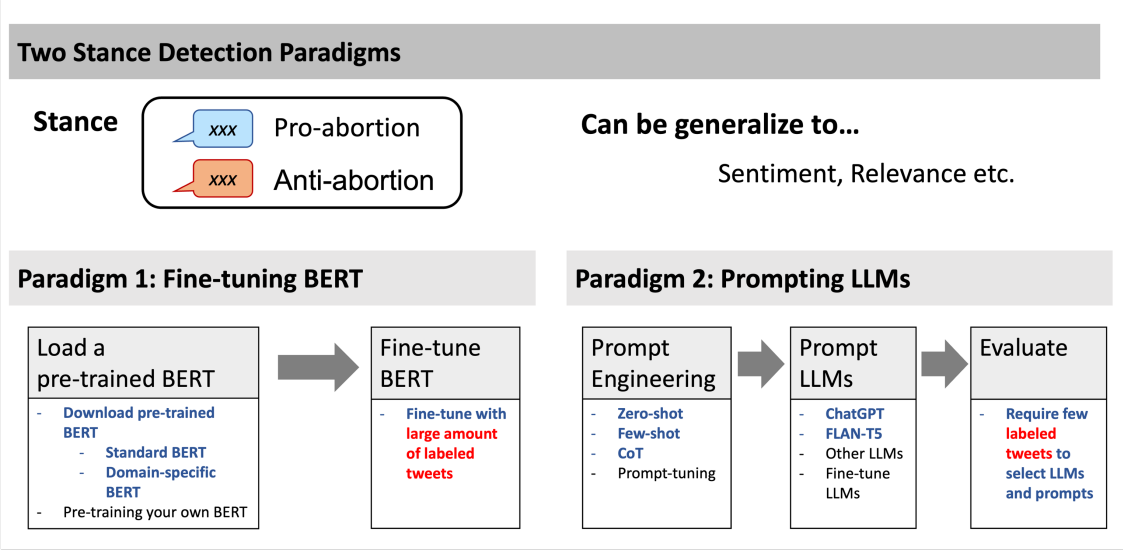
```
[ ]: display_resized_image_in_notebook("dataset_semeval_abortion.png",scale = 0.25)
```

			% of instances in Train				% of instances in Test		
Target	**# total**	**# train**	favor	against	neither	**# test**	favor	against	neither
Data for Task A									
Atheism	733	513	17.9	59.3	22.8	220	14.5	72.7	12.7
Climate Change is Concern	564	395	53.7	3.8	42.5	169	72.8	6.5	20.7
Feminist Movement	949	664	31.6	49.4	19.0	285	20.4	64.2	15.4
Hillary Clinton	984	689	17.1	57.0	25.8	295	15.3	58.3	26.4
Legalization of Abortion	933	653	18.5	54.4	27.1	280	16.4	67.5	16.1
All	4163	2914	25.8	47.9	26.3	1249	24.3	57.3	18.4
Data for Task B									
Donald Trump	707	0	-	-	-	707	20.93	42.29	36.78

Note: The SemEval-2016 dataset contains tweets related to six different topics: Abortion, Atheism, Climate Change, Feminist Movement, Hillary Clinton, and Legalization of Abortion. In this tutorial, we will focus on the Abortion topic only. However, you can easily extend the tutorial to other topics. For an interactive visualization of the SemEval-2016 dataset, please visit here.

1.3 Two Stance Detection Paradigms

```
[ ]: display_resized_image_in_notebook("stance_detection_two_paradigm.png", 0.7)
```



The diagram above illustrates the two paradigms for stance detection: (1) Fine-tuning a BERT model and (2) Prompting Large Language Models (LLMs). The red text highlights the key practical difference between the two approaches, which is the need for large labeled data when fine-tuning a BERT model. The blue texts indicates the parts covered in these two tutorials. While the black parts are not covered in these tutorials, they are important to consider when applying these two paradigms in practice.

In this tutorial, we are exploring two different paradigms for stance detection: 1) fine-tuning a BERT model, and 2) prompting large language models (LLMs) like ChatGPT.

Fine-tuning a BERT model involves training the model on a specific task using a labeled dataset, which adapts the model's pre-existing knowledge to the nuances of the task. This approach can yield strong performance but typically requires a substantial amount of labeled data for the target task.

On the other hand, prompting LLMs involves crafting carefully designed input prompts that guide the model to generate desired outputs based on its pre-trained knowledge. This method does not require additional training, thus significantly reducing the amount of labeled data needed. Note that some labeled data is still required to evaluate the performance.

In this first tutorial, we will focus on the first paradigm: fine-tuning a BERT model, including domain-specific BERT which may be more suitable for our task. In the second tutorial, we will explore the second paradigm: prompting LLMs.

2 Paradigm 1: using BERT for stance detection

In this section, I will briefly introduce BERT, a powerful NLP model that has been widely used in many NLP tasks. we will explain what BERT is, how it is trained, and how it can be used for stance detection. we will also show you how to fine-tune BERT for stance detection using python.

2.1 What is BERT and how it works

BERT, which stands for ***B**idirectional **E**ncoder **R**epresentations from **T**ransformers*, is a groundbreaking natural language processing (NLP) model that has taken the world by storm. Created by researchers at google in 2018, BERT is designed to learn useful representations for words from unlabeled text, which can then be tailored, or, "fine-tuned" for a wide range of NLP tasks, such as stance detection, sentiment analysis, question-answering, among many.

> Note: "Unlabeled text" means that the text does not have any labels, such as the sentiment, or stance, of a tweet. This is in contrast to supervised learning, where the training data is labeled. In supervised learning, the model learns to predict the labels of the text of the training data. When pre-training BERT on unlabeled data, it learns to predict the randomly masked out words in a sentence (explained in details below).

In a nutshell, BERT is a powerful NLP model that leverages 1) the transformer architecture and 2) the pre-training and fine-tuning approach. we will explain these two concepts in more details below.

> Note: In this tutorial, my primary focus is on applying NLP models for stance detection, and I won't be elabortating all the details of BERT. If you're interested in learning more about BERT, I highly recommend checking out the excellent interactive tutorial available at http://jalammar.github.io/illustrated-bert/. This tutorial provides a thorough and visually engaging explanation of BERT's inner workings. Some of the plots in my tutorial are borrowed from this resource.

2.1.1 Bidirectional Context: Understanding Context in Both Directions

Language is complex, and understanding it is no simple task. Traditional NLP models (e.g., RNN; no worries if you don't know what RNN is) have focused on reading text in one direction (e.g.,from left-to-right), making it difficult for them to grasp the full context when trying to understand a word. BERT, however, is designed to process text in both directions, allowing it to understand the meaning of words based on the words that come before and after them.

To explain how this is possible, we first need to understand what a transformer is, and specifically, the critical "self-attention mechanism" component that makes it possible for BERT to understand context in both directions.

2.1.2 A Powerful Backbone Architecture: Transformers with Self-Attention Mechanism

BERT is built upon the **transformer architecture**, the critical backbone of many state-of-the-art NLP models (including both BERT and the LLMs described in the second tutorial), was introduced by Vaswani et al. in their 2017 paper, "Attention Is All You Need."

The key component of the architecture is the "**self-attention mechanism**", which helps the model identify important parts of the input text and understand the relationships between words.

Let's use the concrete example below to illustrate the self-attention mechanism.

[ ]:
```
display_resized_image_in_notebook("example_sentence_self_attention.png",0.2)
```

In this example, what does "**it**" refer to? Does it refer to the animal or the street?

As humans, we understand that "it" refers to the "animal". However, for a machine, determining the correct reference is not a simple task, especially given that the word "street" is closer to "it" than "animal" in the sentence. A naive machine might assume that "it" refers to the "street" because the word "street" is closer to "it" than "animal".

We, as humans, know that "it" refers to the "animal" because we understand that animals can get tired while streets cannot. We also recognize that being too tired is a legitimate reason for not crossing the street. In summary, we can comprehend the meaning of the word "it" by taking into account other words in the sentence, or, in technical terms, the "context".

With the help of the self-attention mechanism, a transformer model takes into account of the "**context**" of a word to understand its meaning.

Let's use a diagram to show how this works. The figure below visualizes how this work. On the left-hand side, the sentence is the input to the self-attention mechanism, while on the right-hand side, the output is also the same sentence (hence the name "self-attention"). The lines between the input and the output dipicts the "attention weight" of each word. In this example, there are two "attention heads", the green one and the orange one. Each head represents a different way of understanding the meaning of the word "it".

Let's focus on the green one ("Head 1") now. This attention head has a high weight on the word "tired", which means that the attention weight of the word "tired" is higher than other words when the model is trying to understand the meaning of the word "it".

[ ]:
```
display_resized_image_in_notebook("self_attention_head_1.png",0.3)
```

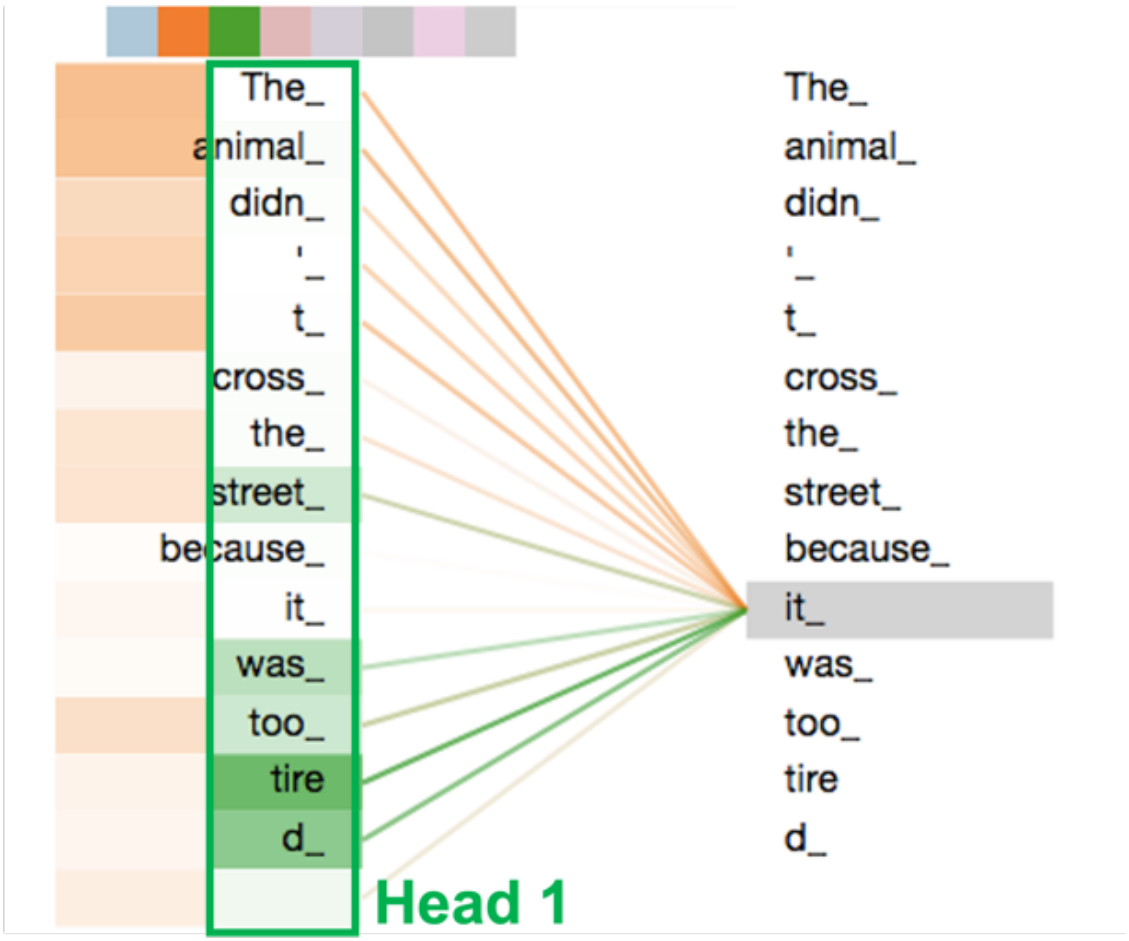


Image modified from: http://jalammar.github.io/illustrated-transformer/

Let's now focus on the orange one ("Head 2"). This attention head has a high weight on the word "animal", indicating that this attention head cares more about the word "animal" when trying to understand the meaning of the word "it".

[ ]:
```
display_resized_image_in_notebook("self_attention_head_2.png",0.3)
```

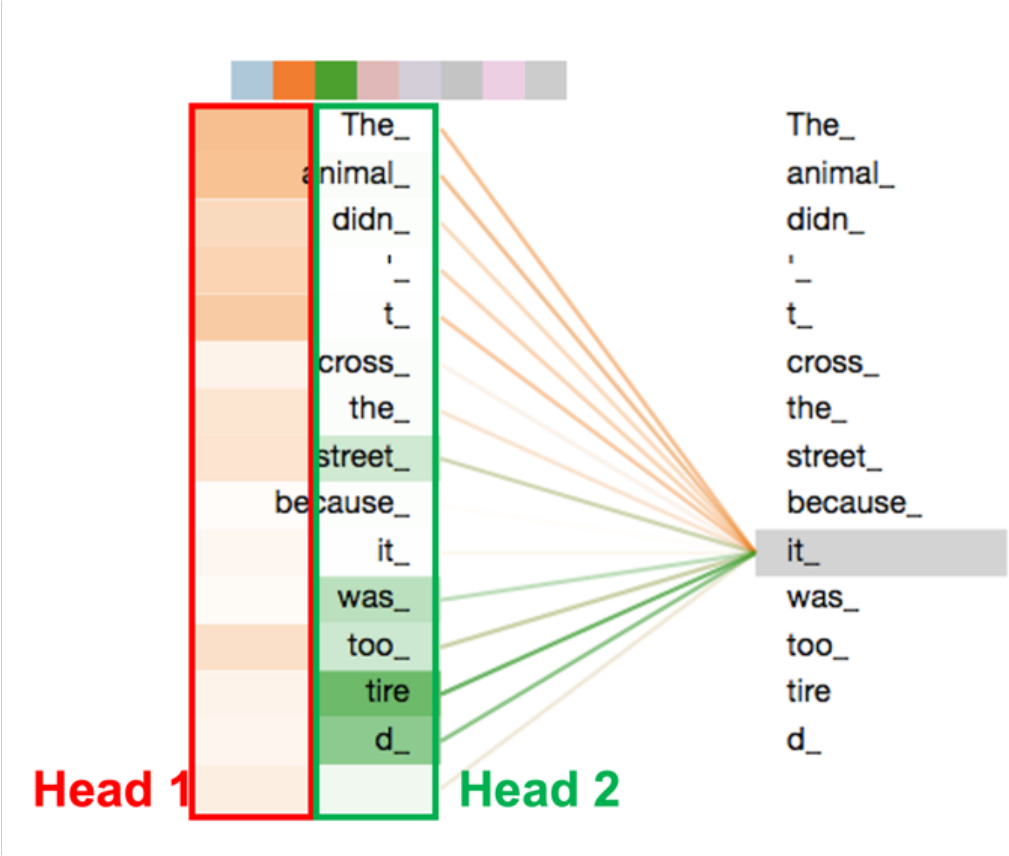


Image modified from: http://jalammar.github.io/illustrated-transformer/

In the actual BERT model, there are 12 attention heads, meaning that the model has 12 different ways of understanding the meaning of any word in a sentence. After we combine the outputs of all 12 attention heads, we then get the representation of the word "it" in the sentence after this "multi-head attention layer". This multi-head attention layer, along with other components (as shown below in the graph below), is called an "encoder".

In the BERT model, for any given input sentence, this attention mechanism is repeated 12 times (i.e., 12 encoders). Intuitively speaking, every time the vector goes through an encoder, it learns a more "abstract" relationship between words in a sentence.

The final product after these 12 layers is the "**representation**" of the input sentence. In total, there are about 110 million trainable parameters in the BERT model.

To make a prediction (e.g., the stance) based on the **representation**, this representation vector is then fed into a linear layer to produce the final output of the model. In the case of BERT, the representation is a vector of 768 numbers (the "hidden units").

> Note: the number of layers, the number of attention heads, the number of encoders, are based on the `bert-base` model, which is the smaller variant of BERT. The larger variant, the `bert-large` model, has 24 encoders, 16 attention heads, and 1024 hidden units, amounting to about 340 million trainable parameters. In this tutorial, we will be

using the `bert-base` model.

```
[ ]: display_resized_image_in_notebook("bert_model_architecture.png",0.7)
```

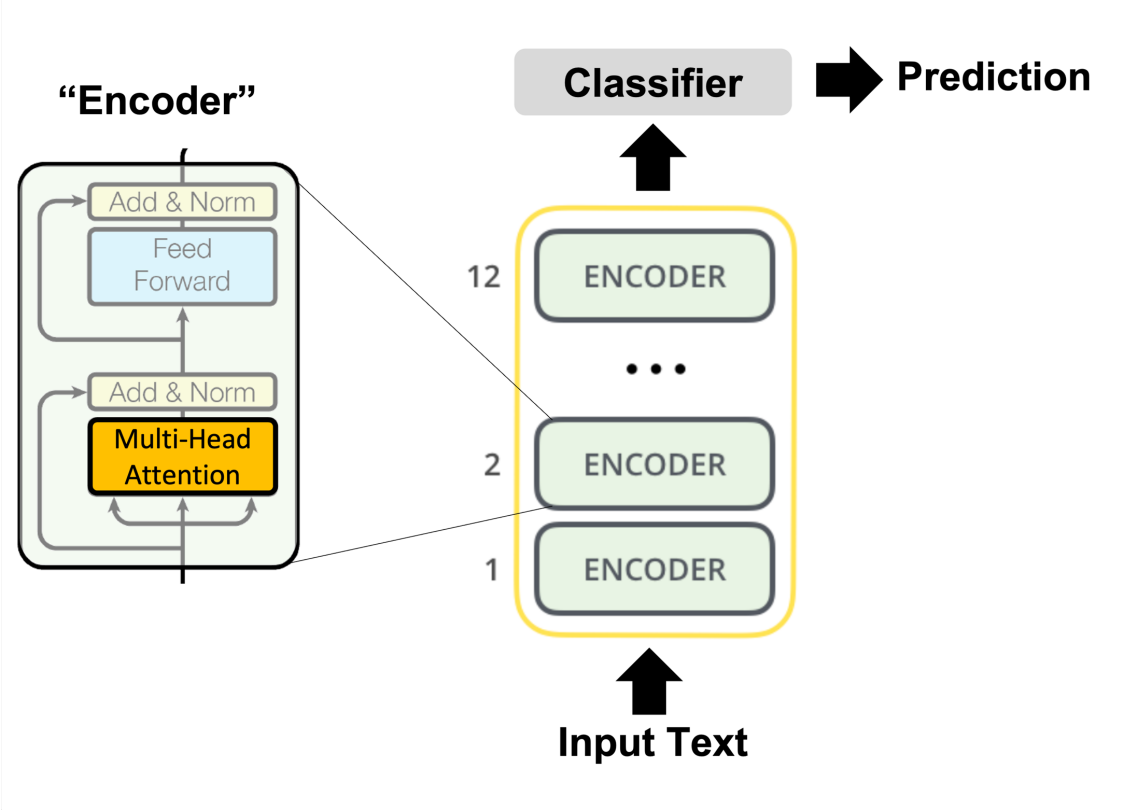


Image modified from: http://jalammar.github.io/illustrated-bert/

Note: The actual self-attention mechanism is more complicated. The "attention weight" is computed by three trainable matrices - the query, key, and value matrices.

Likewise, although the self-attention layers are arguably the most critical component, it is not the only component in a transformer. As shown in the figure aboe, there are other building blocks like layer normalization, residual connection, linear layers, positional encodings etc. If you are interested in learning more about transformers in detail, I highly recommend checking out the interactive tutorial on transformers (by Jay Alammar, the same author of the BERT tutorial linked above): http://jalammar.github.io/illustrated-transformer/. This tutorial provides a comprehensive and visually engaging explanation of the transformer architecture. Some plots in my tutorial are borrowed from this resource. Also note that there are different variants of BERT with different sizes of the transformer architecture. For example, BERT-Base has 12 self-attention layers, while BERT-Large has 24 self-attention layers. In this tutorial, we will be using BERT-Base as a running example.

2.1.3 Pre-training and Fine-tuning: Learning from Lots of Text and Adapting to Specific Tasks

Now we know the architecture of BERT, which is a transformer model with 12 self-attention layers. But how does BERT learn to understand the meaning of words? And how can we use BERT to solve specific NLP tasks, say, stance detection?

Note that the BERT model contains around 110 million parameters, necessitating a substantial amount of data for training. So, how can we effectively train BERT when dealing with a specific task that has a limited dataset? For instance, the Abortion dataset we used in this tutorial comprises only 933 labeled tweets.

One of the key secrets behind BERT's success is its ability to 1) learn from vast amounts of "unlabled text" and then 2) adapt that knowledge to specific tasks with labels. These two components correspond to the two stages when training a BERT model: 1) pre-training and 2) fine-tuning.

1) Pre-training phase During the initial pre-training phase, BERT is exposed to massive amounts of unlabeled text (the raw text itself without any annotation about sentiment, stance etc.). The standard BERT model was pretrained on the entire English Wikipedia and 11k+ online books, which in total contains about 3.3B words.

Why do we want to pre-train BERT on these corpora, even though they are not related to the specific tasks we want to solve (i.e., the Abortion tweet dataset)? The answer is that the pre-training phase allows BERT to learn the general language understanding, for example, the meaning of words, the relationships between words, and the context of words.

[ ]:
```
display_resized_image_in_notebook("bert_pretrain.png",0.5)
```

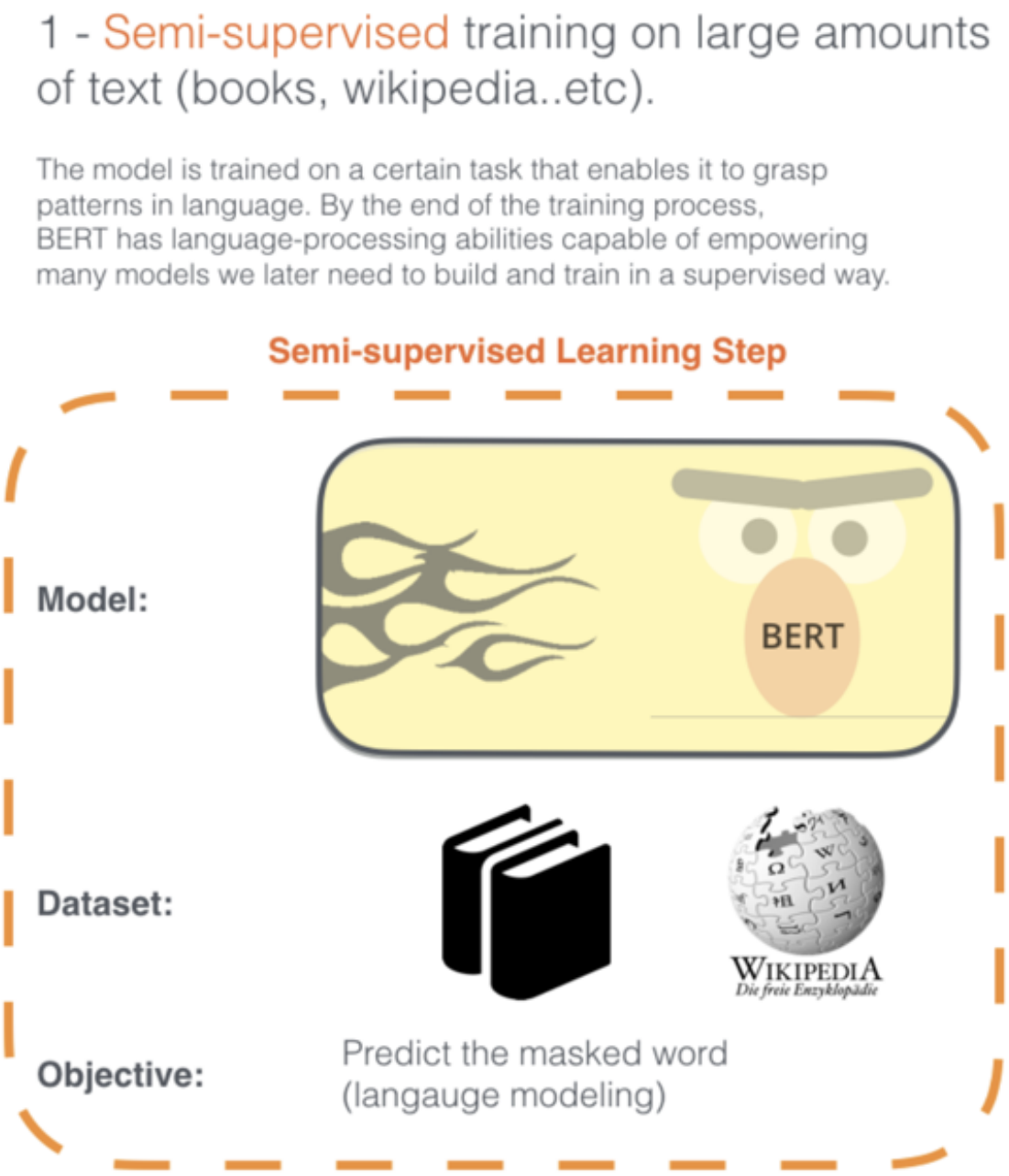


Image modified from: http://jalammar.github.io/illustrated-bert/

In order to learn these general language understanding, in the pre-training phase, BERT uses two different tasks: 1) masked language modeling and 2) next sentence prediction. This phase allows BERT to learn the relationships between words even without any task-specific labels (e.g., stance labels are not needed for pre-training).

The masked language modeling task is a simple task where BERT is asked to predict some "masked out" word in a sentence. For example, given the sentence "The animal didn't cross the street because

it was too tired", when pre-training BERT, the word "it" may be masked out and the model is asked to predict this missing word.

[ ]:
```
display_resized_image_in_notebook("bert_pretrain_mlm.png",0.7)
```

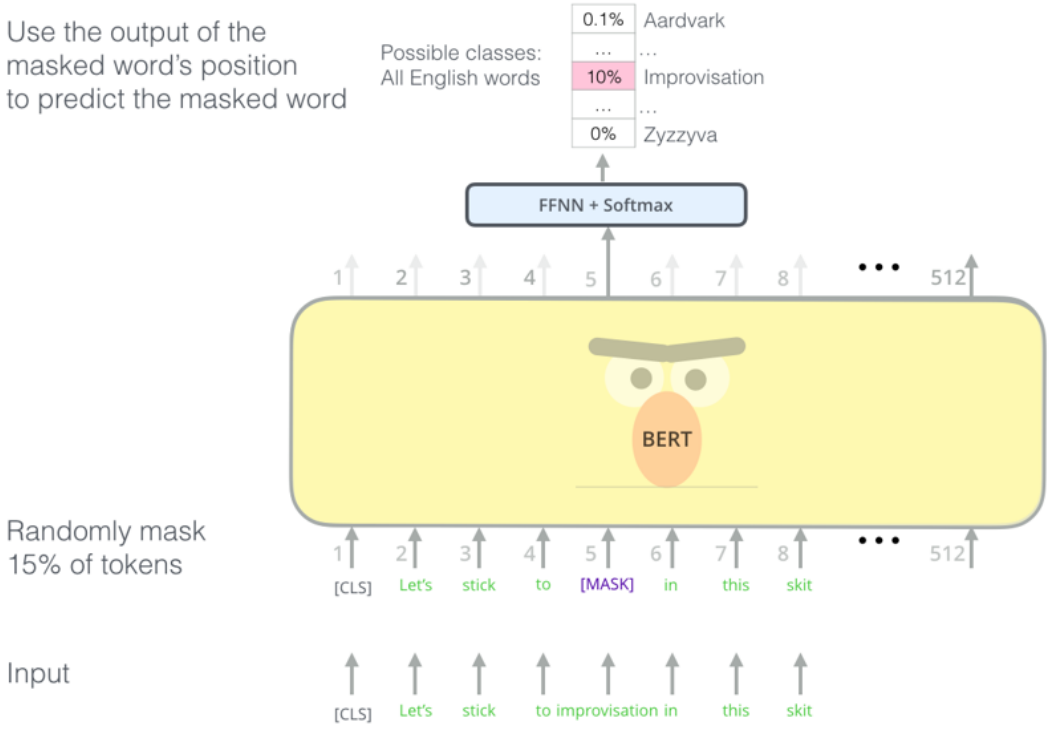


Image modified from: http://jalammar.github.io/illustrated-bert/

The next sentence prediction task is a binary classification task where BERT is asked to predict whether the second sentence is a continuation of the first sentence. For example, if there is a paragraph in the training data, where "*It was an sleepy dog.*" is the second sentence that follows the first sentence "*The animal didn't cross the street because it was too tired.*", then we say that the second sentence is a continuation of the first sentence.

In this task, BERT is asked to decide whether a random sentence is a continuation of another sentence.

[ ]:
```
display_resized_image_in_notebook("bert_pretrain_next_sentence_prediction.png",0.
↪7)
```

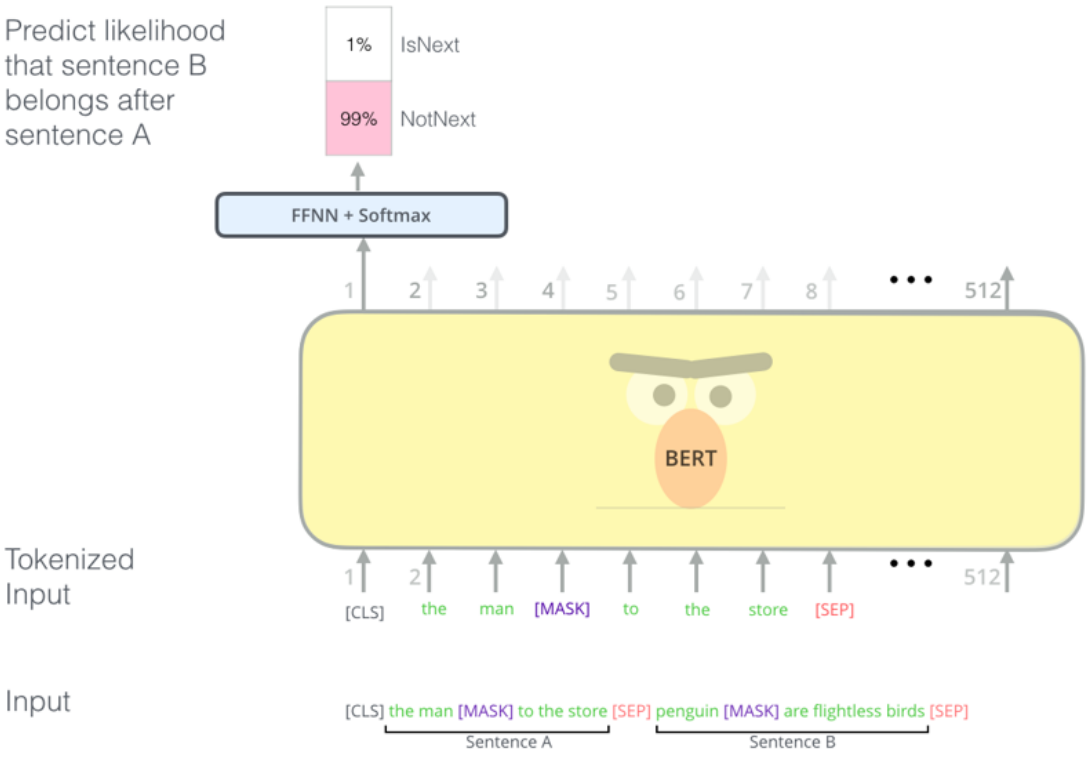


Image modified from: http://jalammar.github.io/illustrated-bert/

Note: One caveat about pre-training is that, the more similar the pre-training corpus is to the task-specific corpus, the better the performance of BERT. For example, if you want to use BERT to solve a stance detection task on tweets about abortion, it is better to pre-train BERT on a corpus that is similar to this dataset. For example, you can pre-train BERT on a corpus that contains tweets (rather than the original Wikipedia and online books corpus). This makes sense because the style of tweets is different from the style of Wikipedia and online books (e.g., they are shorter and more informal). More about this in the next section Considering More Appropriate Pre-trained Models.

2) Fine-tuning phase After pre-training, BERT can be fine-tuned for a specific task with a smaller labeled dataset (e.g., the Abortion tweet dataset). Fine-tuning involves updating the model's weights using the labeled data, allowing BERT to adapt its general language understanding to the specific task. This process is relatively fast and requires less training data compared to training a model from scratch.

[ ]:
```
display_resized_image_in_notebook("bert_fine_tune_stance.png",0.35)
```

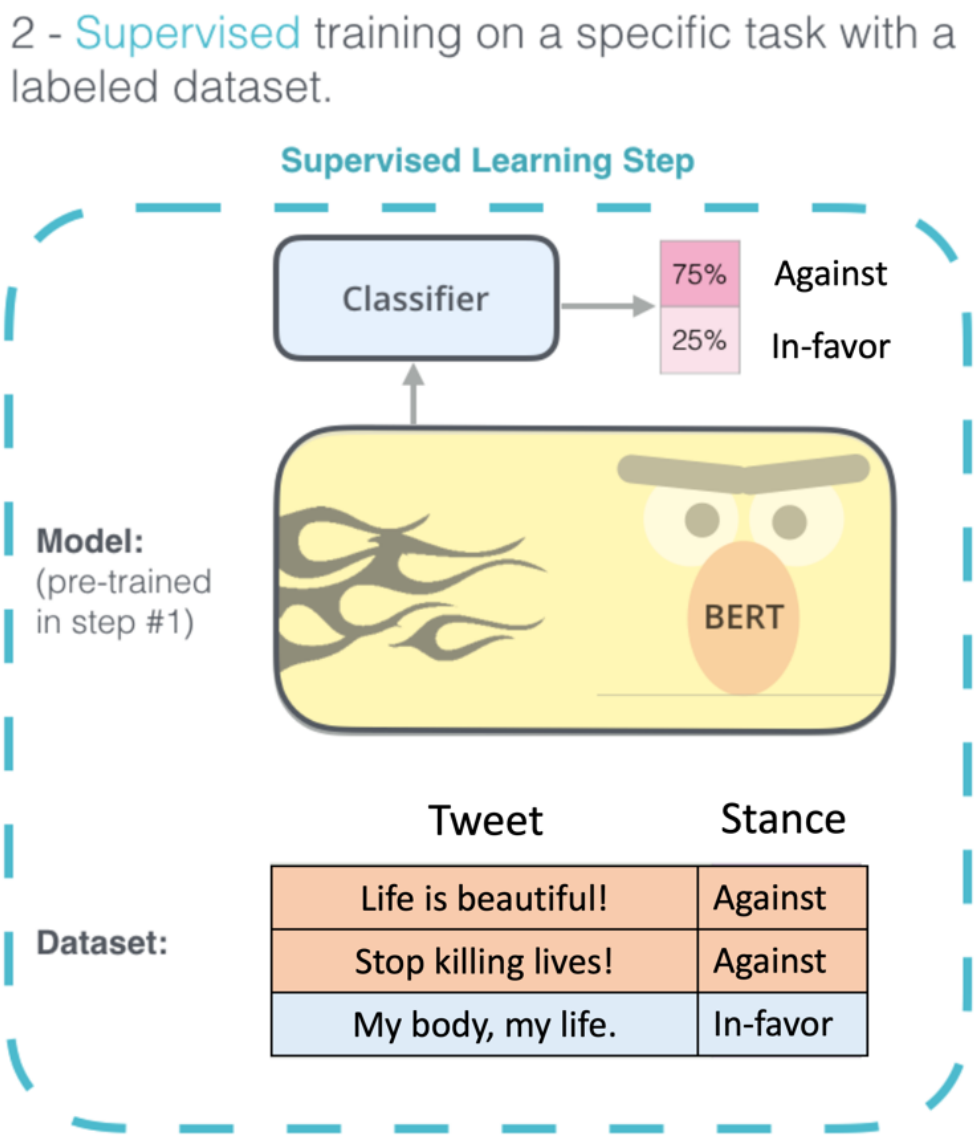


Image modified from: http://jalammar.github.io/illustrated-bert/

2.1.4 BERT's Sub-word Tokenization

One caveat of BERT is that it requires a special "subword-tokenization" process (i.e., WordPiece tokenization). That is, it does not directly encode each individual word, but rather encode each word as a sequence of "sub-word tokens". For example, the word "university" can be broken down into the subwords "uni" and "versity," which are more likely to appear in the corpus than the word "university" itself. This process of breaking down words into subwords is called sub-word tokenization.

Sub-word tokenization is important for several reasons. Just to name two important ones:

Consistent Representation of Similar Words Tokenization ensures that the text is represented in a consistent manner, making it easier for the model to learn and identify patterns in the data. By breaking the text into tokens, the model can focus on the essential units of meaning, allowing it to better understand and analyze the input. For an example, let us consider the following two words that are commonly used in the abortion debate.: "**pro-life**" and "**pro-choice**".

Tokenization can help standardize the text by breaking them down into smaller, overlapping tokens, i.e., `["pro", "-", "life"]` and `["pro", "-", "choice"]`.

By representing the words as a sequence of tokens, the model can more effectively identify the commonality between them (the shared "pro-" prefix) while also distinguishing the unique parts ("life" and "choice"). This approach helps the model learn the relationships between word parts and the context (i.e., other words in the sentence) in a more generalizable way. For example, sub-word tokenization enable the model handle out-of-vocabulary words more effectively, as we will see below.

Handling Out-of-Vocabulary Words One of the challenges in NLP is dealing with words that the model has not encountered during training, also known as out-of-vocabulary (OOV) words. By using tokenization, BERT can handle OOV words more effectively.

For example, suppose we have a sentence containing a relatively newly-coined word: "pro-birth".

Here, the word "**pro-birth**" is a neologism that may not be present in the model's vocabulary during pre-training, particularly if the model was trained on older data. If we used a simple word-based tokenization, the model would struggle to understand this word. However, using a subword tokenization approach, the word can be broken down into smaller parts that the model has likely seen before:

```
["pro", "-", "birth"]
```

This breakdown allows the model to infer the meaning of the previously unseen word based on the subword components it has encountered during training. The model can recognize the "pro" prefix and the suffix "birth". This enables BERT to better understand these out-of-vocabulary words, especially those that are relatively new or coined, making it more robust and adaptable to a wide range of text inputs.

---

3 Programming Exercise: Fine-tuning a BERT Model with HuggingFace

Now, let's fine-tune a standard BERT model using the HuggingFace Transformers library.

> Hugging Face, often called the "GitHub" for NLP models, provides an extensive open-source Transformers library and a model hub, making it easy to access, share, and implement state-of-the-art NLP models like BERT (and other open-source LLMs, more on this in the second tutorial).

First, you need to decide whether you want to train the models on your own or use the predictions I made and uploaded to my GitHub repo. If you're running this notebook for the first time, I recommend setting `DO_TRAIN_MODELS = False` (the default setting below) to save time. This will load the precomputed predictions from my GitHub repo.

However, if you want to try training the models yourself, which I encourage, you can set `DO_TRAIN_MODELS = True` and rerun the notebook. If you're running this on Google Colab, ensure you're using the GPU runtime for a more efficient and faster experience. To enable this, go to `Runtime` -> `Change runtime type` -> `Hardware accelerator` -> `GPU`. Note that even with the GPU runtime, running this entire notebook on Colab will take about 10-15 minutes.

> Note: I have attempted to minimize randomness in the notebook by using a random seed. However, the results you obtain may still vary slightly from those in this notebook due to factors such as different library versions or hardware configurations. To obtain the exact same results as presented in this notebook, keep `DO_TRAIN_MODELS = False` and use the precomputed predictions I made.

[ ]:
```
DO_TRAIN_MODELS = False
```

3.1 Read the Raw Data

[ ]:
```
%load_ext autoreload
%autoreload 2

from data_processor import SemEvalDataProcessor
from utils import get_parameters_for_dataset, tidy_name,␣
↪display_resized_image_in_notebook
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-packages/tqdm/auto.py:21:
TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

[ ]:
```
# set up
SEED = 42
TOPIC_OF_INTEREST = "Abortion"
DATASET = "SEM_EVAL"
par = get_parameters_for_dataset(DATASET)
```

[ ]:
```
# read the raw data
sem_eval_data = SemEvalDataProcessor()
df_raw_train = sem_eval_data.
↪_read_raw_data(read_train=True,read_test=False,topic=TOPIC_OF_INTEREST)
df_raw_test = sem_eval_data.
↪_read_raw_data(read_train=False,read_test=True,topic=TOPIC_OF_INTEREST)
```

Let's look at the raw data first. The raw data is in the format below.

Each line contains a ID of the text, a target topic (which is "Legalization of Abortion"), the raw tweet content, and a stance label (i.e., `FAVOR`, `AGAINST`, `NONE`).

[ ]:
```
df_raw_train[["ID","Target","Tweet","Stance"]].head()
```

[ ]:
```
        ID                   Target  \
2211  2312  Legalization of Abortion
2212  2313  Legalization of Abortion
2213  2314  Legalization of Abortion
2214  2315  Legalization of Abortion
2215  2316  Legalization of Abortion

                                                  Tweet   Stance
2211  I really don't understand how some people are ...  AGAINST
2212  Let's agree that it's not ok to kill a 7lbs ba...  AGAINST
2213  @glennbeck I would like to see poll: How many ...  AGAINST
2214  Democrats are always AGAINST "Personhood" or w...  AGAINST
2215  @CultureShifting "If you don't draw the line w...     NONE
```

[ ]:
```
print("number of tweets in the training data: ",len(df_raw_train))
```

```
number of tweets in the training data:  603
```

The testing data below has the same format. Note that this set is not used for training, but for evaluating a trained model's performance on unseen data.

[ ]:
```
df_raw_test[["ID","Target","Tweet","Stance"]].head()
```

[ ]:
```
        ID                   Target  \
969   10970  Legalization of Abortion
970   10971  Legalization of Abortion
971   10972  Legalization of Abortion
972   10973  Legalization of Abortion
973   10974  Legalization of Abortion

                                                 Tweet   Stance
969  Need a ProLife R.E. Agent? - Support a ProLife...  AGAINST
970  Where is the childcare program @joanburton whi...  AGAINST
971  I get several requests with petitions to save ...  AGAINST
972  we must always see others as Christ sees us,we...  AGAINST
973  PRAYERS FOR BABIES Urgent prayer one in Lexing...  AGAINST
```

[ ]:
```
print("number of tweets in the test data: ",len(df_raw_test))
```

```
number of tweets in the test data:  280
```

Let's look at some examples of the raw tweets.

[ ]:
```
# convert to list
df_raw_train[["Tweet"]].values[[7,21]].tolist()
```

[ ]:
```
[['RT @createdequalorg: "We\'re all human, aren\'t we? Every human life is worth
the same, and worth saving." -J.K. Rowling #... #SemST'],
 ['Follow #Patriot --> @Enuffis2Much.  Thanks for following back!!  #Truth
```

```
#Liberty #Justice #ProIsrael #WakeUpAmerica #FreeAmirNow #SemST']]
```

We can see that the tweets are not very clean.

For example, the first tweet contains a retweet tag (i.e., "RT @createdequalorg"). This tag entails that the tweet is a retweet of another tweet. This messsage is not part of the content of the original tweet, and thus should be removed.

Aside from the retweet tag, the tweets also contain some other noise, such as some special characters (e.g., /'). We will also remove these special characters.

In addition, the mentions (e.g., "@Enuffis2Much") contains the reference to other users. These mentions may confuse the model and should be removed as well.

> Note: In practice, these non-language features can be leveraged to improve the model's performance for various text sources. However, we will not be exploring that approach in this tutorial to maintain a general focus on the core techniques and to accommodate a wide range of text types.

Finally, all tweets end with a special hashtag (e.g., "#SemST"). These hashtags are added by the owners of the SemST dataset to indicate the stance of the tweet, and are not part of the original tweet content. We will also remove these special hashtags.

3.2 Preprocess the Raw Data

We will preprocess the raw data to address the issues mentioned above.

Aside from preprocessing the raw tweets, we will also partition the training data into a training set and a validation set (with a 4:1 ratio). The validation set will be used to evaluate the model's performance during training.

[ ]:
```
# preprocess the raw tweets
sem_eval_data.preprocess()
df_processed = sem_eval_data._read_preprocessed_data(topic=TOPIC_OF_INTEREST).
 ↪reset_index(drop=True)
```

[ ]:
```
# partition the data into train, vali, test sets
df_partitions = sem_eval_data.partition_processed_data(seed=SEED,verbose=False)
```

Let's look at the preprocessed data. The first thing to notice is that there is a new column called "partitions". This column indicates whether the tweet belongs to the training set, validation set, or testing set.

[ ]:
```
df_processed.head()
```

[ ]:

	ID	tweet	topic	label	\
0	2312	i really don't understand how some people are ...	Abortion	AGAINST	
1	2313	let's agree that it's not ok to kill a 7lbs ba...	Abortion	AGAINST	
2	2314	@USERNAME i would like to see poll: how many a...	Abortion	AGAINST	
3	2315	democrats are always against 'personhood' or w...	Abortion	AGAINST	
4	2316	@USERNAME 'if you don't draw the line where i'...	Abortion	NONE	

```
  partition
0     train
1     train
2     train
3     train
4     train
```

Let's look at the preprocessed tweets.

We can see that the tweets are now much cleaner. For example, the retweet tag, special characters, and special hashtags have been removed. The mention tags are replaced by a sentinel token (i.e., "@USERNAME").

[ ]:
```
df_processed[["tweet"]].values[[7,21]].tolist()
```

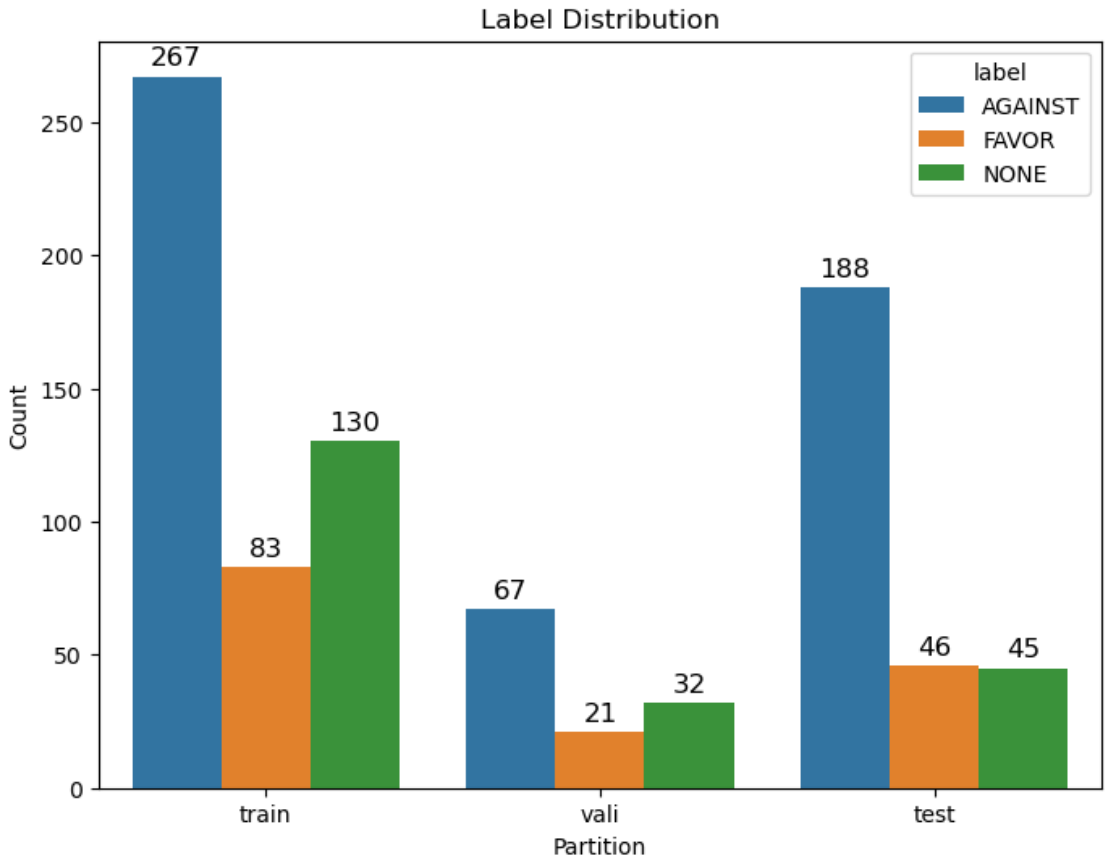
[ ]:
```
[["'we're all human, aren't we? every human life is worth the same, and worth
saving.' -j.k. rowling #..."],
 ['follow #patriot --> @USERNAME. thanks for following back!! #truth #liberty
#justice #proisrael #wakeupamerica #freeamirnow']]
```

Let's look at the distribution of the stance labels across the training, validation, and testing sets.

[ ]:
```
# add a "count" column to count the number of tweets in each partition
df_label_dist = df_partitions[df_partitions.topic == TOPIC_OF_INTEREST].
↪value_counts(['partition','label']).sort_index()
```

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

df_label_dist_plot = df_label_dist.reset_index()


# Create the bar plot
plt.figure(figsize=(8, 6))
ax = sns.barplot(data=df_label_dist_plot, x="partition", y=0, hue="label",␣
↪order=["train", "vali", "test"])

# Customize the plot
plt.xlabel("Partition")
plt.ylabel("Count")
plt.title("Label Distribution")

# Add count on top of each bar
for p in ax.patches:
    ax.annotate(
        f'{p.get_height():.0f}',
        (p.get_x() + p.get_width() / 2., p.get_height()),
```

```
        ha='center',
        va='baseline',
        fontsize=12,
        color='black',
        xytext=(0, 5),
        textcoords='offset points'
    )
# Show the plot
plt.show()
```



4 Train a stadard BERT model

- Here, I use the BERT-base-uncased model, which is a standard BERT model with 12 self-attention layers and 110 million parameters.

[ ]:
```
import pandas as pd
from os.path import join

from transformers import TrainingArguments
```

```
from transformers import Trainer
from transformers import AutoTokenizer
from transformers import AutoModelForSequenceClassification
import torch as th


from data_processor import SemEvalDataProcessor
from utils import process_dataframe, preprocess_dataset,␣
↪partition_and_resample_df, evaluate_trained_trainer_over_sets,␣
↪func_compute_metrics_sem_eval, remove_saved_models_in_checkpoint,␣
↪remove_checkpoint_dir, seed_all,convert_stance_code_to_text
```

4.1 Set up

[ ]:
```
# use the standard bert model
ENCODER = "bert-base-uncased"
PATH_OUTPUT_ROOT = par.PATH_RESULT_SEM_EVAL_TUNING
```

Use GPU if available

[ ]:
```
device = th.device('cuda' if th.cuda.is_available() else 'cpu')
print('Using device:', device)
seed_all(SEED)
```

```
Using device: cuda
```

[ ]:
```
# specify the output path
path_run_this = join(PATH_OUTPUT_ROOT, ENCODER)
file_metrics = join(path_run_this, "metrics.csv")
file_confusion_matrix = join(path_run_this, "confusion_matrix.csv")
file_predictions = join(path_run_this, "predictions.csv")
print("path_run_this:", path_run_this)
print("file_metrics:", file_metrics)
print("file_confusion_matrix:", file_confusion_matrix)
print("file_predictions:", file_predictions)
```

```
path_run_this:
/home/sean/prelim_stance_detection/results/semeval_2016/tuning/bert-base-uncased
file_metrics:
/home/sean/prelim_stance_detection/results/semeval_2016/tuning/bert-base-
uncased/metrics.csv
file_confusion_matrix:
/home/sean/prelim_stance_detection/results/semeval_2016/tuning/bert-base-
uncased/confusion_matrix.csv
file_predictions:
/home/sean/prelim_stance_detection/results/semeval_2016/tuning/bert-base-
uncased/predictions.csv
```

[ ]:
```
# Load the preprocessed data and the partitions
data_processor = SemEvalDataProcessor()
func_compute_metrics = func_compute_metrics_sem_eval()
file_processed = data_processor.
 ↪_get_file_processed_default(topic=TOPIC_OF_INTEREST)
df_partitions = data_processor.read_partitions(topic=TOPIC_OF_INTEREST)

df = process_dataframe(input_csv=file_processed,
                       dataset=DATASET)
```

Note that there is a label imbalance issue in the dataset. There are way more tweets with the `AGAINST` label than the other two labels. This may cause the model to be biased towards predicting the `AGAINST` label.

To address this issue, we will first upsample the training set to make the number of tweets with each label equal.

> To learn more about the data imbalance issue, I recommend taking a look at this tutorial https://towardsdatascience.com/5-techniques-to-work-with-imbalanced-data-in-machine-learning-80836d45d30c

[ ]:
```
# upsample the minority class
dict_df = partition_and_resample_df(df, SEED, "single_domain",
                                    factor_upsample=1,
                                    read_partition_from_df=True,
                                    df_partitions=df_partitions)
```

Let's check if the upsampled data is balanced now.

[ ]:
```
# before upsampling
dict_df["train_raw"]["label"].apply(lambda x: convert_stance_code_to_text(x,␣
 ↪DATASET)).value_counts().sort_index()
```

[ ]:
```
AGAINST    267
FAVOR       83
NONE       130
Name: label, dtype: int64
```

[ ]:
```
# after upsampling
dict_df["train_upsampled"]["label"].apply(lambda x:␣
 ↪convert_stance_code_to_text(x, DATASET)).value_counts().sort_index()
```

[ ]:
```
AGAINST    267
FAVOR      267
NONE       267
Name: label, dtype: int64
```

4.2 Load the tokenizer and tokenize the tweets

Recall that BERT requires a special "subword-tokenization" process (i.e., WordPiece tokenization). That is, it does not directly encode each individual word, but rather encode each word as a sequence of "sub-word tokens". For example, the word "university" can be broken down into the subwords "uni" and "versity," which are more likely to appear in the corpus than the word "university" itself. This process of breaking down words into subwords is called sub-word tokenization.

[ ]:
```
# load the tokenizer and preprocess the data
tokenizer = AutoTokenizer.from_pretrained(ENCODER)
dict_dataset = dict()
for data_set in dict_df:
    dict_dataset[data_set] = preprocess_dataset(dict_df[data_set],
                                                tokenizer,
                                                keep_tweet_id=True,
                                                col_name_tweet_id=par.TEXT_ID)
```

[ ]:
```
# the processed data set has the following structure
# - text: the text of each tweet
# - label: the label of each stance
# - input_ids: the "token ids" of each tweet
dict_dataset["train_upsampled"]
```

[ ]:
```
Dataset({
    features: ['text', 'ID', 'label', '__index_level_0__', 'input_ids',
'token_type_ids', 'attention_mask'],
    num_rows: 801
})
```

4.2.1 Let's look at one example tweet after tokenization

The sentence "*i really don't understand how some people are pro-choice. a life is a life no matter if it's 2 weeks old or 20 years old.*" is converted into the following tokens ID:

[101, 1045, 2428, 2123, 1005, 1056, 3305, 2129, 2070, 2111, 2024, 4013, 1011, 3601, 1012, 1037, 2166, 2003, 1037, 2166, 2053, 3043, 2065, 2009, 1005, 1055, 1016, 3134, 2214, 2030, 2322, 2086, 2214, 1012, 102, 0, 0, . . . , 0]

> The "0" at the end are the padding tokens. They not used to train the model. Rather, they are used to make all the tweets within a batch have the same length. This is a common practice when training neural network models using batches.

[ ]:
```
print("The original text of this tweet: \n {}\n".
 ↪format(dict_dataset["train_upsampled"]["text"][0]))
print("The label of this tweet: \n {}\n".format(convert_stance_code_to_text(
    dict_dataset["train_upsampled"]["label"][0].item(), DATASET)))
```

```
print("The token ids of this tweet: \n {}\n".
 ↪format(dict_dataset["train_upsampled"]["input_ids"][0]))
```

```
The original text of this tweet:
 i really don't understand how some people are pro-choice. a life is a life no
matter if it's 2 weeks old or 20 years old.

The label of this tweet:
 AGAINST

The token ids of this tweet:
 tensor([ 101, 1045, 2428, 2123, 1005, 1056, 3305, 2129, 2070, 2111, 2024, 4013,
        1011, 3601, 1012, 1037, 2166, 2003, 1037, 2166, 2053, 3043, 2065, 2009,
        1005, 1055, 1016, 3134, 2214, 2030, 2322, 2086, 2214, 1012,  102,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0])
```

The tokens ID can be converted back to the original tokens, also using the tokenizer.

Let look at the first 10 tokens of the first tweet.

Notice that the word "pro-choice" is broken down into the subwords "pro", "-", and "choice", as explained above.

[ ]:
```
for i in range(15):
    token_id_this = dict_dataset["train_upsampled"]["input_ids"][0][i].item()
    token_this = tokenizer.decode(token_id_this)
    print("token_id: {}; token: {}".format(
        token_id_this,
        token_this)
    )
```

```
token_id: 101; token: [CLS]
token_id: 1045; token: i
token_id: 2428; token: really
token_id: 2123; token: don
token_id: 1005; token: '
token_id: 1056; token: t
token_id: 3305; token: understand
token_id: 2129; token: how
token_id: 2070; token: some
token_id: 2111; token: people
token_id: 2024; token: are
token_id: 4013; token: pro
token_id: 1011; token: -
token_id: 3601; token: choice
token_id: 1012; token: .
```

The "[CLS]" token is another special token (just like the padding token aboe) that is added to the beginning of each tweet. It is how BERT knows that the tweet is the beginning of a new sentence.

[ ]:
```
# "decode" the entire sequence
tokenizer.
 ↪decode(dict_dataset["train_upsampled"]["input_ids"][0],skip_special_tokens=True)
```

[ ]:
```
"i really don't understand how some people are pro - choice. a life is a life no
matter if it's 2 weeks old or 20 years old."
```

Next, we want to load a pre-trained BERT model, which will be used to initialize the weights of our model. We will use `bert-base-uncased` model, which is a standard BERT model with 12 self-attention layers and 110 million parameters.

[ ]:
```
print("The BERT model to use: {}".format(ENCODER))
```

```
The BERT model to use: bert-base-uncased
```

[ ]:
```
# load the pretrained model
model = AutoModelForSequenceClassification.from_pretrained(ENCODER,
                                                           num_labels = par.
 ↪DICT_NUM_CLASS[DATASET])
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertForSequenceClassification:
['cls.predictions.transform.LayerNorm.weight', 'cls.seq_relationship.bias',
'cls.predictions.transform.LayerNorm.bias',
'cls.predictions.transform.dense.bias',
'cls.predictions.transform.dense.weight', 'cls.predictions.decoder.weight',
'cls.seq_relationship.weight', 'cls.predictions.bias']
- This IS expected if you are initializing BertForSequenceClassification from
the checkpoint of a model trained on another task or with another architecture
```

```
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing BertForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at bert-base-uncased and are newly initialized:
['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

Great! After loading the pre-trained BERT model, now we are ready to fine-tune the BERT model. We are going to use the classes `Trainer` and `TrainingArguments` provided by the HuggingFace library.

Let's specify the training arguments, including the number of epochs, the batch size, and the learning rate etc.

While the model is being trained, we retain the best model at each epoch based on the macro F1 score on the validation set. The macro F1 score is the average of the F1 scores across all three stance classes.

> To learn more about macro-F1 score, I recommend taking a look at this tutorial https://towardsdatascience.com/micro-macro-weighted-averages-of-f1-score-clearly-explained-b603420b292f#:~:text=The%20macro%2Daveraged%20F1%20score,regardless%20of%20their%20s

[ ]:
```python
# specify the training arguments
training_args = TrainingArguments(
    # dir to save the model checkpoints
    output_dir=path_run_this,
    # how often to evaluate the model on the eval set
    # - logs the metrics on vali set
    evaluation_strategy="epoch",
    # how often to log the training process to tensorboard
    # - only log the train loss , lr, epoch etc info and not the metrics
    logging_strategy="epoch",
    # how often to save the model on the eval set
    # - load_best_model_at_end requires the save and eval strategy to match
    save_strategy="epoch",
    # limit the total amount of checkpoints. deletes the older checkpoints in␣
 ↪output_dir.
    save_total_limit=1,
    # initial learning rate for the adamw optimizer
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=6,
    weight_decay=0.01,
```

```
    seed=SEED,
    data_seed=SEED,
    # retain the best model (evaluated by the metric on the eval set)
    load_best_model_at_end=True,
    metric_for_best_model="f1_macro",
    # number of updates steps to accumulate the gradients for, before performing␣
 ↪a backward/update pass
    gradient_accumulation_steps=1,
    optim="adamw_torch",
    report_to="none"
)
```

[ ]:
```
# Specify the trainer
trainer = Trainer(model=model, args=training_args,
                  train_dataset=dict_dataset["train_upsampled"],
                  eval_dataset=dict_dataset["vali_raw"],
                  compute_metrics=func_compute_metrics
                  )
```

4.2.2 Fine-tune the BERT model!

[ ]:
```
# Train the model
if DO_TRAIN_MODELS:
  trainer.train()
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/home/sean/prelim_stance_detection/scripts/utils.py:667: FutureWarning:
load_metric is deprecated and will be removed in the next major version of
datasets. Use 'evaluate.load' instead, from the new library Hugging Face Evaluate:
https://huggingface.co/docs/evaluate
  metric_computer[name_metric] = load_metric(name_metric)
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

4.2.3 Evaluate the model performance

[ ]:
```
%%capture --no-stderr
if DO_TRAIN_MODELS:
    # evaluate on each partition
    df_metrics, df_confusion_matrix, dict_predictions = \
        evaluate_trained_trainer_over_sets(trainer,
                                           DATASET,
                                           dict_dataset, "set",
                                           return_predicted_labels=True,
                                           keep_tweet_id=True,
                                           col_name_tweet_id=par.TEXT_ID)
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
```

```
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

[ ]:
```python
if DO_TRAIN_MODELS:
  # save the evaluation results
  df_metrics.to_csv(file_metrics)

  # save the confusion matrix
  df_confusion_matrix.to_csv(file_confusion_matrix)

  # save the predictions
  # concatenate the predictions
  # - "train_raw", "val_raw", "test_raw"
  df_predictions = pd.concat([dict_predictions[key] for key in ["train_raw",␣
↪"vali_raw", "test_raw"]], axis=0)
  # save only the tweet_id and the predicted label
  df_predictions = df_predictions[["ID", "predicted_label"]]
  # convert the predicted label to the stance code
  df_predictions["predicted_label"] = df_predictions["predicted_label"].
↪apply(lambda x: convert_stance_code_to_text(x, DATASET))
  df_predictions.to_csv(file_predictions, index=False)

  # remove the saved model
  remove_saved_models_in_checkpoint(path_run_this)
  # remove the checkpoints
  remove_checkpoint_dir(path_run_this)
```

---

5 Considering More Appropriate Pre-trained Domain-specific Models

While the `bert-base-uncased` model serves as a good starting point, there are other pre-trained models specifically designed for social media text analysis that use more relevant pre-training data. Two such models are BERTweet and polibertweet-mlm.

As mentioned earlier, when aiming to use BERT for a stance detection task on tweets about abortion, it is more effective to pre-train BERT on a corpus that is more similar to the Abortion tweet

dataset. Given that tweets are generally shorter and more informal than Wikipedia and online books, it makes sense to pre-train BERT on a corpus that primarily consists of tweets (rather than the original Wikipedia and online books corpus). This is precisely the approach taken by domain-specific models like `BERTweet` and `polibertweet-mlm`, which focus on capturing the nuances and characteristics of social media text, making them better suited for stance detection tasks in this context.

> Note for advanced readers: It is important to mention that both BERTweet and polibertweet-mlm are based on the roberta-base architecture, a variant of BERT that has been optimized for improved performance. As a result, these domain-specific models not only benefit from more appropriate pre-training data but also from the enhancements offered by the roberta-base architecture. To understand the differences between BERT and roberta-base, I recommend taking a look at this tutorial

5.1 BERTweet

`BERTweet` is a pre-trained model specifically designed for processing and understanding Twitter data. It is trained on a large corpus of 850 million English tweets. As Twitter text contains unique language patterns, slang, and abbreviations, BERTweet is expected to perform better on stance detection tasks involving tweets compared to the general-purpose BERT model. For more details, see: https://huggingface.co/docs/transformers/model_doc/bertweet

5.2 Polibertweet-mlm

`Polibertweet-mlm` is a pre-trained model specifically designed for Twitter data, with a focus on political discourse. The dataset used for pretraining contains over 83 million English tweets related to the 2020 US Presidential Election.

As the Abortion stance dataset may also involve political topics, polibertweet-mlm can also to be a suitable model for this stance detection task. For more details, see: https://huggingface.co/kornosk/polibertweet-political-twitter-roberta-mlm

5.2.1 Helper Function

Before we proceed with fine-tuning the models, we first create a wrapper function for each type of pretrained model. Using a wrapper function for the training pipeline streamlines the process of experimenting with different models, ensuring consistency, reproducibility, and maintainability.

[ ]:
```
def train(encoder):
    # specify the output paths
    encoder_name_tidy = tidy_name(encoder)
    path_run_this = join(PATH_OUTPUT_ROOT, encoder_name_tidy)
    file_metrics = join(path_run_this, "metrics.csv")
    file_confusion_matrix = join(path_run_this, "confusion_matrix.csv")
    file_predictions = join(path_run_this, "predictions.csv")

    # Load the preprocessed data and the partitions
    df = process_dataframe(input_csv=file_processed, dataset=DATASET)
    # upsample the minority class
    dict_df = partition_and_resample_df(df, SEED, "single_domain",
```

```
                                        factor_upsample=1,
                                        read_partition_from_df=True,
                                        df_partitions=df_partitions)
    # load the tokenizer and preprocess the data
    tokenizer = AutoTokenizer.from_pretrained(encoder)
    dict_dataset = dict()
    for data_set in dict_df:
        dict_dataset[data_set] = preprocess_dataset(dict_df[data_set],
                                                    tokenizer,
                                                    keep_tweet_id=True,
                                                    col_name_tweet_id=par.
↪TEXT_ID)
    # specify the training arguments
    training_args = TrainingArguments(
        # dir to save the model checkpoints
        output_dir=path_run_this,
        evaluation_strategy="epoch",
        logging_strategy="epoch",
        save_strategy="epoch",
        save_total_limit=1,
        learning_rate=2e-5,
        per_device_train_batch_size=16,
        per_device_eval_batch_size=16,
        num_train_epochs=6,
        weight_decay=0.01,
        seed=SEED,
        data_seed=SEED,
        load_best_model_at_end=True,
        metric_for_best_model="f1_macro",
        gradient_accumulation_steps=1,
        optim="adamw_torch",
        report_to="none"
    )

    # load the pretrained model
    model = AutoModelForSequenceClassification.from_pretrained(encoder,
                                                               num_labels=par.
↪DICT_NUM_CLASS[DATASET])

    # specify the trainer
    trainer = Trainer(model=model, args=training_args,
                      train_dataset=dict_dataset["train_upsampled"],
                      eval_dataset=dict_dataset["vali_raw"],
                      compute_metrics=func_compute_metrics
                      )

    # train the model
```

```
    trainer.train()

    # evaluate on each partition
    df_metrics, df_confusion_matrix, dict_predictions = \
        evaluate_trained_trainer_over_sets(trainer,
                                           DATASET,
                                           dict_dataset, "set",
                                           return_predicted_labels=True,
                                           keep_tweet_id=True,
                                           col_name_tweet_id=par.TEXT_ID)
    # save the evaluation results
    df_metrics.to_csv(file_metrics)

    # save the confusion matrix
    df_confusion_matrix.to_csv(file_confusion_matrix)

    # save the predictions
    # concatenate the predictions
    # - "train_raw", "val_raw", "test_raw"
    df_predictions = pd.concat([dict_predictions[key] for key in ["train_raw",␣
↪"vali_raw", "test_raw"]], axis=0)
    # save only the tweet_id and the predicted label
    df_predictions = df_predictions[["ID", "predicted_label"]]
    # convert the predicted label to the stance code
    df_predictions["predicted_label"] = df_predictions["predicted_label"].
↪apply(lambda x: convert_stance_code_to_text(x, DATASET))
    df_predictions.to_csv(file_predictions, index=False)

    # remove the saved model
    remove_saved_models_in_checkpoint(path_run_this)
    # remove the checkpoints
    remove_checkpoint_dir(path_run_this)
```

[ ]:
```
if DO_TRAIN_MODELS:
  # BERTweet, having the same architecture as BERT-base (Devlin et al., 2019),␣
↪is trained using the RoBERTa pre-training procedure
  # - https://huggingface.co/docs/transformers/model_doc/bertweet
  train("vinai/bertweet-base")
```

```
Special tokens have been added in the vocabulary, make sure the associated word
embeddings are fine-tuned or trained.
Some weights of the model checkpoint at vinai/bertweet-base were not used when
initializing RobertaForSequenceClassification: ['roberta.pooler.dense.weight',
'lm_head.layer_norm.bias', 'lm_head.layer_norm.weight', 'lm_head.dense.bias',
'lm_head.decoder.bias', 'lm_head.dense.weight', 'roberta.pooler.dense.bias',
'lm_head.decoder.weight', 'lm_head.bias']
- This IS expected if you are initializing RobertaForSequenceClassification from
```

```
the checkpoint of a model trained on another task or with another architecture
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing RobertaForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of RobertaForSequenceClassification were not initialized from the
model checkpoint at vinai/bertweet-base and are newly initialized:
['classifier.out_proj.bias', 'classifier.dense.weight',
'classifier.out_proj.weight', 'classifier.dense.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
```

```
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

<IPython.core.display.HTML object>

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

<IPython.core.display.HTML object>

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

<IPython.core.display.HTML object>

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

<IPython.core.display.HTML object>

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

<IPython.core.display.HTML object>

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

<IPython.core.display.HTML object>

[ ]:
```
if DO_TRAIN_MODELS:
    # polibertweet-mlm
    # - https://huggingface.co/kornosk/polibertweet-political-twitter-roberta-mlm
    train("kornosk/polibertweet-mlm")
```

```
Some weights of the model checkpoint at kornosk/polibertweet-mlm were not used
when initializing RobertaForSequenceClassification: ['lm_head.layer_norm.bias',
'lm_head.layer_norm.weight', 'lm_head.dense.bias', 'lm_head.decoder.bias',
```

```
'lm_head.dense.weight', 'lm_head.decoder.weight', 'lm_head.bias']
- This IS expected if you are initializing RobertaForSequenceClassification from
the checkpoint of a model trained on another task or with another architecture
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing RobertaForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of RobertaForSequenceClassification were not initialized from the
model checkpoint at kornosk/polibertweet-mlm and are newly initialized:
['classifier.out_proj.bias', 'classifier.dense.weight',
'classifier.out_proj.weight', 'classifier.dense.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
```

```
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

```
/home/sean/miniconda3/envs/prelim/lib/python3.10/site-
packages/torch/nn/parallel/_functions.py:68: UserWarning: Was asked to gather
along dimension 0, but all input tensors were scalars; will instead unsqueeze
and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
```

```
<IPython.core.display.HTML object>
```

6 Compare the performance of the different models

[ ]:
```
from result_summarizer import ResultSummarizer
```

[ ]:
```
LIST_MODEL_NAME =␣
↪["bert-base-uncased","vinai_bertweet_base","kornosk_polibertweet_mlm"]
# summarize the results
# - bert-base-uncased
result_summarizer = ResultSummarizer(dataset=DATASET,
                                     list_version_output=LIST_MODEL_NAME,
                                     eval_mode="single_domain",
                                     model_type="single_domain_baseline",
                                     task=None,
                                     file_name_metrics="metrics.csv",
                                     file_name_confusion_mat="confusion_matrix.
↪csv",
                                     path_input_root=PATH_OUTPUT_ROOT,
                                     path_output=join(PATH_OUTPUT_ROOT,␣
↪"summary"))
# write the summary to a csv file
df_hightlight_metrics = result_summarizer.
↪write_hightlight_metrics_to_summary_csv(
                        list_metrics_highlight=['f1_macro', 'f1_NONE',␣
↪'f1_FAVOR', 'f1_AGAINST'],
                        list_sets_highlight=['train_raw', 'vali_raw',␣
↪'test_raw'],
                        col_name_set="set")
# reorder the rows
df_hightlight_metrics['version'] = pd.
↪Categorical(df_hightlight_metrics['version'], categories=LIST_MODEL_NAME,␣
↪ordered=True)
df_hightlight_metrics = df_hightlight_metrics.sort_values('version').
↪reset_index(drop=True)

# visualize the results and save the figures
plot_con_mat = result_summarizer.visualize_confusion_metrices_over_domains_comb(
    ["train_raw", "vali_raw", "test_raw"],
    preserve_order_list_sets=True)
```

```
<Figure size 1500x750 with 0 Axes>
```

```
<Figure size 1500x750 with 0 Axes>
```

```
<Figure size 1500x750 with 0 Axes>
```

[ ]:
```
# print the summary table
df_hightlight_metrics[["version","set","f1_macro","f1_NONE","f1_FAVOR","f1_AGAINST"]][df_hightl
↪set.isin(["test_raw"])]
```

[ ]:

	version	set	f1_macro	f1_NONE	f1_FAVOR	f1_AGAINST
2	bert-base-uncased	test_raw	0.4748	0.4196	0.4275	0.5775
5	vinai_bertweet_base	test_raw	0.5797	0.5323	0.5401	0.6667

8	kornosk_polibertweet_mlm	test_raw	0.5616	0.5440	0.4762	0.6645

In this tutorial, we compared the performance of three different BERT models for stance detection: `bert-base-uncased`, `vinai_bertweet_base`, and `kornosk_polibertweet_mlm`. The latter two models are domain-specific, designed specifically for tweets. The results show that both domain-specific models outperform the general `bert-base-uncased` model in terms of macro-F1 scores. The `vinai_bertweet_base` model achieves the best performance with an macro-F1 score of 0.5797, followed by `kornosk_polibertweet_mlm` with a score of 0.5616, and finally, the `bert-base-uncased` model with a score of 0.4748. This demonstrates the advantage of using domain-specific models when dealing with tasks that involve specific types of data, such as social media text.

6.1 Analyzing the Confusion Matrix for Deeper Insights

Given the performance differences observed among the three models, it is valuable to investigate their "confusion matrices" for each model Examining these matrices can also help identify potential biases, challenges, and opportunities for improvement.

> Note: Examining the confusion matrix is essential because it provides a detailed overview of the model's performance across different classes. It reveals not only the correct predictions (true positives) but also the instances where the model made errors (false positives and false negatives). By analyzing the confusion matrix, we can identify patterns in misclassifications and gain insights into the strengths and weaknesses of the model. Here is a great tutorial on how to interpret the confusion matrix and its relationships with macro-F1 scores: https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826

Below are the confusion matrices for the standard `bert-base-uncased` model.

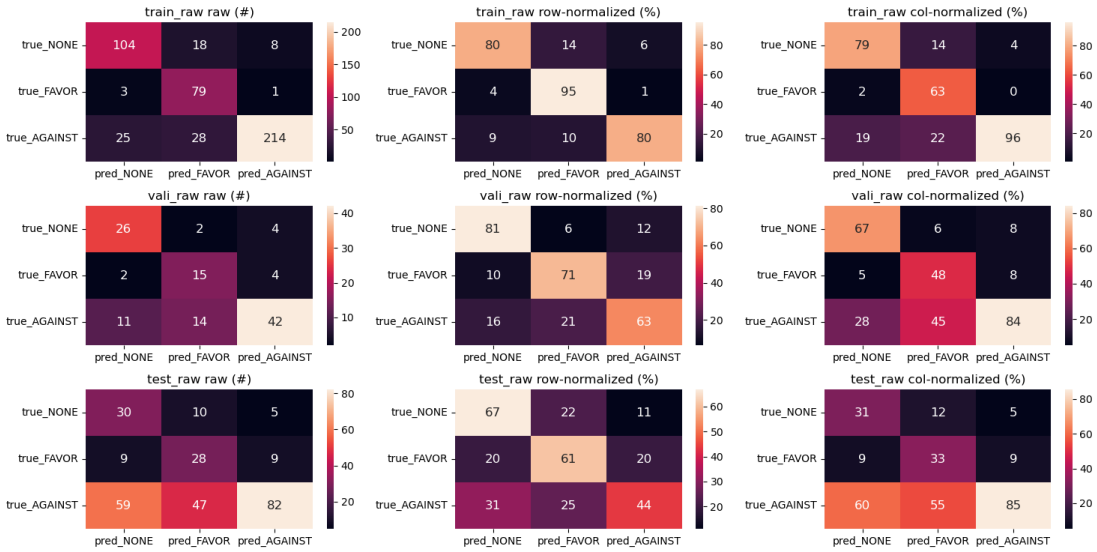
The second row are the matrices for the training set, the second row are for the validation set, and the third row are for the test set.

Each row of matrices consists of three types:

1. The leftmost matrices are the raw confusion matrices.
2. The middle matrices show the confusion matrices normalized by row (i.e., the sum of each row equals 100). In these matrices, the diagonal values correspond to the recall value of each class.
3. The rightmost matrices illustrate the confusion matrices normalized by column (i.e., the sum of each column equals 100). In these matrices, the diagonal values are the precision value for each class.

In each matrix, the rows represent the true labels, and the columns represent the predicted labels. The diagonal elements denote correct predictions, while the off-diagonal elements indicate incorrect predictions.

[ ]:
```
display_resized_image_in_notebook(join(PATH_OUTPUT_ROOT,"summary","bert-base-uncased_comb_confu
↪png"))
```

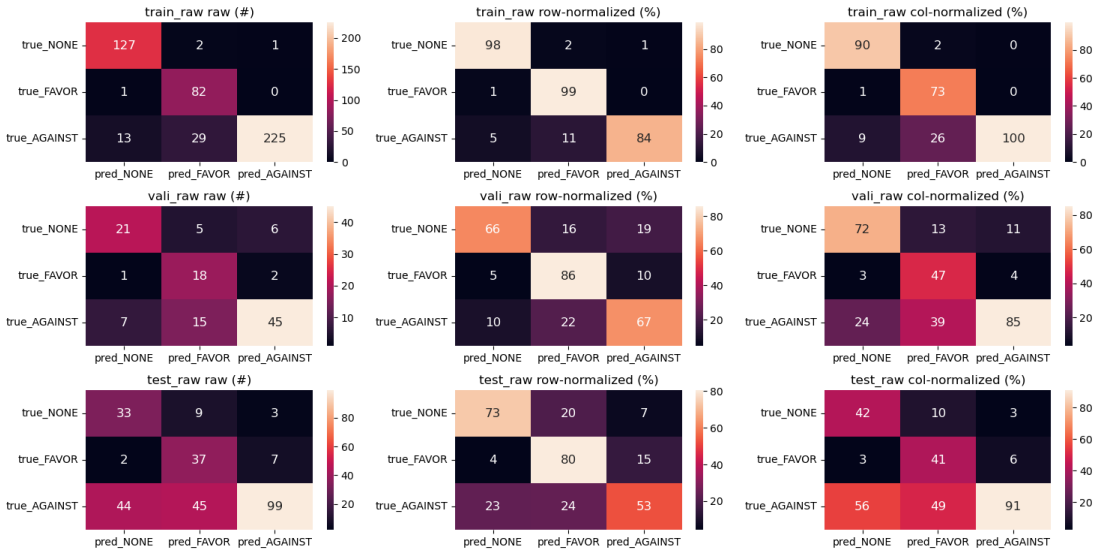


As shown in the test set confusion matrices, the model demonstrates proficiency in distinguishing between the `FAVOR` and `NONE` stances. However, it faces challenges in accurately predicting the `AGAINST` stance, frequently misclassifying them as `NONE`.

Now, let's examine the confusion matrix for the domain-specific `vinai_bertweet_base` model, which has the highest macro-F1 score.

The confusion matrices below reveal that the model is slight better at classifying all three stance classes compared to the standard `bert-base-uncased` model.

[ ]:
```
display_resized_image_in_notebook(join(PATH_OUTPUT_ROOT,"summary","vinai_bertweet_base_comb_con
↪png"))
```



7 Conclusion

In this first part of the tutorial series, we have explored stance detection, its importance, and one of the two distinct paradigms for approaching the task: tuning BERT. We discussed the concept of BERT, its architecture, and the fundamentals of transfer learning with pre-trained models. Additionally, we covered the process of tokenization in BERT, which is crucial for preparing input data.

We then walked through the process of fine-tuning a BERT model using the HuggingFace Transformers library. This involved installing the library, loading a pre-trained BERT model, preparing and processing the labeled dataset, fine-tuning the model for stance detection, and finally evaluating the model and analyzing the results.

Throughout the tutorial, we emphasized the benefits of using domain-specific models for tasks involving specific types of data, such as social media text. In the upcoming second part of the tutorial series, we will cover the second paradigm, prompting LLMs, and demonstrate its effectiveness in stance detection.

---

8 Part 2: Stance Detection on Tweets using Large Language Models (LLMs)

Author: Yun-Shiuan Chuang (yunshiuan.chuang@gmail.com)

Note: This tutorial consists of two separate Python notebooks. This notebook is the second one. The first notebook can be found here. I recommend that you go through the first notebook before the second one as the second notebook builds on top of the first one.

1. First notebook: Fine-tuning BERT models: include standard BERT and domain-specific BERT

- https://colab.research.google.com/drive/1nxziaKStwRnSyOLI6pLNBaAnB_aB6IsE?usp=sharing

2. Second notebook (this one): Prompting large language models (LLMs): include ChatGPT, FLAN-T5 and different prompt types (zero-shot, few-shot, chain-of-thought)

- https://colab.research.google.com/drive/1IFr6Iz1YH9XBWUKcWZyTU-1QtxgYqrmX?usp=sharing

8.1 Getting Started: Overview, Prerequisites, and Setup

Objective of the tutorial: This tutorial will guide you through the process of stance detection on tweets using two main approaches: fine-tuning a BERT model and using large language models (LLMs).

Prerequisites:

- If you want to run the tutorial without editting the codes but want to understand the content
  - Understand what transformer is and what BERT is. If you are not familiar with these concepts, I strongly recommend that you go through the first tutorial here before this one.
  - Basic Python skills: functions, classes, pandas, etc.
  - Basic ML knowledge: train-validation-test split, F1 score, forward pass, backpropagation etc.

Acknowledgements

- While the application of LLMs on stance detection is my own work, some part of this tutorials, e.g., GPT-3, are inspired by the following tutorials. Some of the figures are also modified from the images in these tutorials. I highly recommend you check them out if you want to learn more about LLMs.
  - https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling
  - https://jalammar.github.io/how-gpt3-works-visualizations-animations/
- This tutorial was created with the assistance of ChatGPT (GPT-4), a cutting-edge language model developed by OpenAI. The AI-aided writing process involved an iterative approach, where I provided the model with ideas for each section and GPT-4 transformed those ideas into well-structured paragraphs. Even the outline itself underwent a similar iterative process to refine and improve the tutorial structure. Following this, I fact-checked and revised the generated content, asking GPT-4 to make further revisions based on my evaluation, until I took over and finalized the content.

Setup

1. Before we begin with Google Colab, please ensure that you have selected the GPU runtime. To do this, go to `Runtime` -> `Change runtime type` -> `Hardware accelerator` -> `GPU`. This will ensure that the note will run more efficiently and quickly.
2. Now, let's download the content of this tutorial and install the necessary libraries by running the following cell.

[1]:
```
from os.path import join
ON_COLAB = True
if ON_COLAB:
  !git clone --single-branch --branch colab https://github.com/yunshiuan/
 ↪prelim_stance_detection.git
  !python -m pip install pandas datasets openai tiktoken accelerate transformers␣
 ↪transformers[sentencepiece] torch==1.12.1+cu113 -f https://download.pytorch.
 ↪org/whl/torch_stable.html emoji -q
  %cd /content/prelim_stance_detection/scripts
else:
  # if you are not on colab, you have to set up the environment by yourself. You␣
 ↪would also need a machine with GPU.
  %cd scripts
```

```
Cloning into 'prelim_stance_detection'...
remote: Enumerating objects: 513, done.
remote: Counting objects: 100% (36/36), done.
```

```
remote: Compressing objects: 100% (24/24), done.
remote: Total 513 (delta 21), reused 24 (delta 12), pack-reused 477
Receiving objects: 100% (513/513), 58.56 MiB | 18.24 MiB/s, done.
Resolving deltas: 100% (254/254), done.
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪492.4/492.4
kB 4.4 MB/s eta 0:00:00
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪73.6/73.6 kB
6.3 MB/s eta 0:00:00
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪1.7/1.7 MB
10.9 MB/s eta 0:00:00
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪244.2/244.2
kB 7.5 MB/s eta 0:00:00
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪7.4/7.4 MB
21.6 MB/s eta 0:00:00
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪1.8/1.8 GB
555.8 kB/s eta 0:00:00
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪361.8/361.8 kB
24.3 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪115.3/115.3
kB 9.9 MB/s eta 0:00:00
     ␣
 ↪----------------------------------------------------------------------------------------------------
 ↪212.5/212.5 kB
17.4 MB/s eta 0:00:00
```

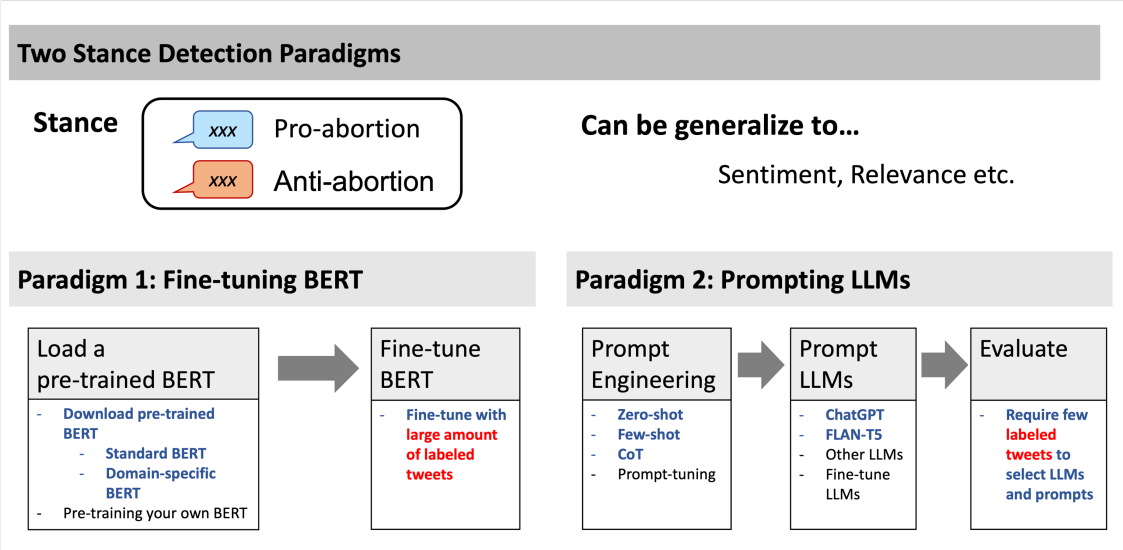
```
     ␣
 ↪---------------------------------------------------------------------------------------
 ↪134.8/134.8 kB
12.4 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------
 ↪268.8/268.8 kB
22.7 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------
 ↪7.8/7.8 MB
43.8 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------
 ↪1.3/1.3 MB
49.7 MB/s eta 0:00:00
     ␣
 ↪---------------------------------------------------------------------------------------
 ↪1.3/1.3 MB
53.6 MB/s eta 0:00:00
  Building wheel for emoji (pyproject.toml) ... done
ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the following
dependency conflicts.
torchaudio 2.0.2+cu118 requires torch==2.0.1, but you have torch 1.12.1+cu113
which is incompatible.
torchdata 0.6.1 requires torch==2.0.1, but you have torch 1.12.1+cu113 which is
incompatible.
torchtext 0.15.2 requires torch==2.0.1, but you have torch 1.12.1+cu113 which is
incompatible.
torchvision 0.15.2+cu118 requires torch==2.0.1, but you have torch 1.12.1+cu113
which is incompatible.
/content/prelim_stance_detection/scripts
```

[2]:
```python
# a helper function to load images in the notebook
from PIL import Image as PILImage
from IPython.display import display, Image

from parameters_meta import ParametersMeta as par
PATH_IMAGES = join(par.PATH_ROOT, "images")
```

```
def display_resized_image_in_notebook(file_image, scale=1,␣
↪use_default_path=True):
    """Display an image in a notebook."""
    # - https://stackoverflow.com/questions/69654877/
↪how-to-set-image-size-to-display-in-ipython-display
    if use_default_path:
        file_image = join(PATH_IMAGES, file_image)
    image = PILImage.open(file_image)
    display(image.resize((int(image.width * scale), int(image.height * scale))))

def display_gif_in_notebook(file_git, use_default_path=True):
    if use_default_path:
        file_git = join(PATH_IMAGES, file_git)
    return Image(file_git)
```

8.2 Two Stance Detection Paradigms

[3]:
```
display_resized_image_in_notebook("stance_detection_two_paradigm.png", 0.7)
```



The diagram above illustrates the two paradigms for stance detection: (1) Fine-tuning a BERT model and (2) Prompting Large Language Models (LLMs). The red text highlights the key practical difference between the two approaches, which is the need for large labeled data when fine-tuning a BERT model. The blue texts indicates the parts covered in these two tutorials. While the black parts are not covered in these tutorials, they are important to consider when applying these two paradigms in practice.

In this tutorial and the previous tutorial, we are exploring two different paradigms for stance

detection: 1) fine-tuning a BERT model, and 2) prompting large language models (LLMs) like ChatGPT.

Fine-tuning a BERT model involves training the model on a specific task using a labeled dataset, which adapts the model's pre-existing knowledge to the nuances of the task. This approach can yield strong performance but typically requires a substantial amount of labeled data for the target task.

On the other hand, prompting LLMs involves crafting carefully designed input prompts that guide the model to generate desired outputs based on its pre-trained knowledge. This method does not require additional training, thus significantly reducing the amount of labeled data needed. Note that some labeled data is still required to evaluate the performance.

In this second tutorial, we will focus on the second paradigm: prompting LLMs. We will explore two classes of LLMs: ChatGPT and FLAN-T5. We will also explore different prompt types: zero-shot, few-shot, and chain-of-thought.

> For an in-depth exploration of the first paradigm, I invite you to refer to my previous tutorial, which can be found here: Fine-tuning BERT for Stance Detection.

---

9 Paradigm 2: Using Large Language Models (LLMs) for Stance Detection

Large Language Models (LLMs) like GPT-3 are gaining significant attention in recent years. These models are designed to understand and generate human-like text by learning from vast amounts of data. In the context of stance detection, LLMs can be used to classify text based on the stance towards a particular topic.

> Strictly speaking, BERT is also a type of LLMs. The "LLMs" covered in this notebook actually refer to a special type of LLMs, the "generative models", which means they can be used to generate text. In contrast, BERT is a discriminative model, which means it can only be used to classify text. I am using the term "LLMs" to refer to these generative models for the sake of simplicity.

Although there are many variants of LLMs, I will explain LLMs with GPT-3 as an example. Later in this tutorial, I will then point out the differences between GPT-3 and other LLMs, such as ChatGPT, FLAN-T5, etc.

Because GPT-3 and BERT are both based on transformers, they share many similarities. I will explain GPT-3 by contrasting it with BERT, assuming that you alreadt know how BERT works, which is covered in the first notebook.

9.1 Contrast GPT-3 with BERT

9.1.1 Model Architecture: Encoder vs. Decoder

Both BERT and GPT-3 are transformer-based models, which means they both employ self-attention layers to learn the relationships between words in a text. However, they utilize self-attention layers differently.

Notably, in their model architecture, BERT uses "encoder blocks," while GPT-3 employs "decoder blocks." Due to this distinction, BERT is often referred to as an "encoder" model, while GPT-3 is commonly known as a "decoder" model.

Encoder (BERT)

In simple terms, an encoder model like BERT encodes an input sequence into a fixed-length vector (after 12 self-attention layers for BERT). This vector, or, representation, is then used to classify the input sequence.

Let's look at a concrete example from the Abortion dataset.

> "*It's so brilliant that #lovewins - now extend the equality to women's rights #abortion-rights*"

Recall from the previous tutorial that when an encoder model like BERT processes a sentence, it utilizes bidirectional context to accurately capture the meaning of the sentence. Afterward, we can fine-tune the model using a labeled dataset to adapt it to our specific task.

Decoder (GPT-3)

On the other hand, a decoder model like GPT-3 are designed to **generate** (rather than encode) a sequence from left to right, one token at a time. If we provide a partially complete sequence of words (also known as "prompt") to GPT-3, it will help complete the sequence (also known as "**conditional text generation**").

So, if we rephrase the same sentence into the following format, and provide GPT-3 with this partially complete sequece (also known is "**prompt**"), we can use GPT-3 to generate its prediction of the stance based on the prompt.

Let's rephrase the sentence into the following "**prompt**":

> "What is the stance of the tweet below with respect to 'Legalization of Abortion'? Please use exactly one word from the following 3 categories to label it: 'in-favor', 'against', 'neutral-or-unclear'. Here is the tweet: '*It's so brilliant that #lovewins - now extend the equality to women's rights #abortionrights.*' The stance of the tweet is:"

With this rephrased sequence, we convert the stance detection task - a classification task, into a text generation task. We can then use GPT-3 to generate the stance of the tweet.

Note that GPT-3, like BERT, still uses self-attention layers to learn the relationships between words. The critical distinction is that when GPT-3 generates a sequence, it can only look at the words before the word to be generated (in this example, the prompt), rather than the bidirectional context like BERT.

The primary difference in the usege of self-attention mechanism is shown in the following figure. On the left, we have the encoder model (BERT), where the self-attention layers are bidirectional, i.e., covering both the left and right context.

On the right, we have the decoder model (GPT-3), where the self-attention layers are unidirectional, i.e., only covering the left context when evaluating the word to predict.

[4]:
```
display_resized_image_in_notebook("encoder_vs_decoder.png",0.3)
```

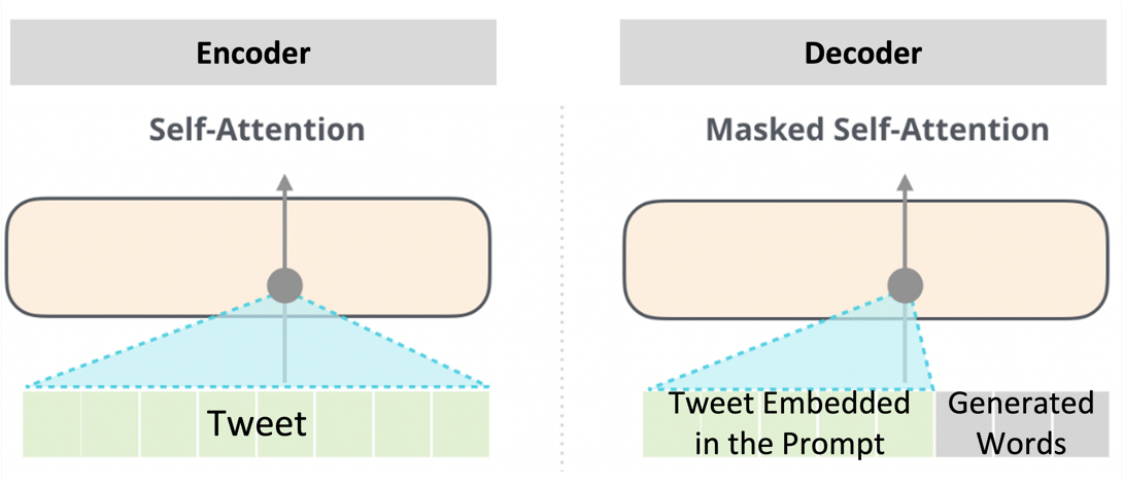


Image modified from: https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Note: Another important difference between BERT and GPT-3 is the size of the context they can handle. BERT uses a context of 512 tokens, while GPT-3 can handle a much larger context depending on the specific model variant, with the largest version handling up to 4097 tokens. This means that GPT-3 can take into account a larger number of words when capturing the meaning of a word, which allows GPT-3 to consider more context surrounding a word and potentially leads to better understanding and generation capabilitie

9.1.2 Different Pre-training Methods

Because BERT and GPT-3 are different types of transformer models (encoder vs. decoder), they use different pre-training methods.

BERT

As mentioned in the previous tutorial, BERT is pre-trained using the 1) masked language modeling task and the 2) next sentence prediction task. These tasks enable the BERT model to learn the relationships between words. However, these tasks alone is not aligned with a classification task like stance detection, which is why we need to fine-tune BERT using a labeled dataset to adapt it to our specific task.

GPT-3

In contrast, GPT-3 is pre-trained using the **unidirectional language modeling task**. In this task, the model is tasked with predicting the next word in a sequence, given the previous words in the sequence.

The animation below shows how GPT-3 is pre-trained using this task. In the example, the partially complete sequence is "a robot must", and the model is tasked with predicting the next word "obey" based on the previous words. This pretraining task enables the GPT-3 model to predict the stance of a given prompt with the need of fine-tuning (more on this later).

[5]:
```
display_gif_in_notebook("gpt3_pretraining.gif")
```

[5]: <IPython.core.display.Image object>

Image modified from: https://jalammar.github.io/how-gpt3-works-visualizations-animations/

9.1.3 Model Size

In addition to the differences in their model architecture, a crucial distinction between BERT and GPT-3 lies in their scale. GPT-3 primarily derives its power from its immense model size and the substantial amount of data utilized for pre-training.

While BERT indeed boasts a massive number of model parameters (hundreds of millions), GPT-3 surpasses it with an even more colossal number of parameters, reaching hundreds of billions!

Below is a table showing the number of parameters for different LLMs. As you can see, GPT-3 has about 500 times more parameters than the largest BERT model.

Here is the reordered table as requested:

Model Variant	Number of Parameters	Transformer Layers	Attention Heads	Hidden Size
BERT-Large	340 million	24	16	1024
GPT-3	175 billion	96	96	12288

There are different variants of GPT-3 and different variants of BERT, with different number of layers, hidden size, and attention heads etc. Here, I am using the largest variant of both models for comparison.

9.1.4 Pre-Traing Data Size

Because GPT-3 has a much larger number of parameters than BERT, it also requires a much larger corpus of text to pre-train.

Specifically, GPT-3 is trained on about **400 billion tokens**, while BERT is trained on about **3.3 billion tokens**. This means that GPT-3 has about access to about 100 times more information than BERT, which can lead to better performance.

- BERT's training data: English Wikipedia + about 11k books = about 2.5 billion English words = 3.3 billion tokens
- GPT-3's training data: English Wikipedia + books + webpages crawled from the internet since 2008 with about 1 trillion words = about 400 billion tokens

Note: Because BERT is trained on English corpus, and GPT-3 is trained on multiple languages, we can't directly compare the number of the words. Instead, we can compare the number of tokens after tokenization. Here, the tokenzier is the Byte Pair Encoding (BPE) method, which is the tokenizer used by GPT-3. If you want to know more about this tokenizer, I recommend this tutorial.: https://huggingface.co/learn/nlp-course/chapter6/5?fw=pt

9.1.5 The necessity of fine-tuning and labeled data

The role of fine-tuning in BERT and GPT-3 differs significantly.

As mentioned earlier, after the pre-training phase, BERT requires fine-tuning on specific tasks of interest (such as stance detection) using labeled data. The reason for this is that BERT is pre-trained on a masked language modeling task and a next sentence prediction task, which are not directly related to stance detection.

In contrast, GPT-3 is pre-trained on the unidirectional language modeling task, which enables it to directly generate a prediction for any given task of interest. In our use case, this means that using ahe GPT-3 model for stance detection task does not necessarily require fine-tuning with labeled data. This implication is huge as collecting labeled data is often a time-consuming and expensive process.

> Note: While it is not necessary for the model to perform well on a given task without fine-tuning on labeled data (see the prompting section below), OpenAI does offer the option to fine-tune the GPT-3 model using labeled data, Note that, however, fine-tining the GPT-3 model and using a fine-tuned model can be expensive. Please see the OpenAI's pricing page for more details.

The Potential of Prompting Because GPT-3 model is pre-trained in a way to predict the next word given a sequence (the "prompt"), it matter critically what the actual prompt is.

For instance, the example earlier demonstrates a "zero-shot prompt", which asks GPT-3 for stance prediction without providing any example tweets.

> "What is the stance of the tweet below with respect to 'Legalization of Abortion'? Please use exactly one word from the following 3 categories to label it: 'in-favor', 'against', 'neutral-or-unclear'. Here is the tweet: '*It's so brilliant that #lovewins - now extend the equality to women's rights #abortionrights.*' The stance of the tweet is:"

The name zero-shot prompt comes from the fact that no example tweets are given, and GPT-3 is expected to generate the stance based on the prompt alone.

Prompt choice significantly impacts the model's performance, an I will dedicate a separate section below to discuss different types of prompts and their implications.

9.1.6 Conclusion: GPT-3 vs. BERT on stance detection

In summary, BERT and GPT-3 are both powerful transformer-based models, but with significant differences in their architecture, pre-training methods, model size, and use of fine-tuning.

BERT, an encoder model, relies on bidirectional context and fine-tuning to perform well on specific tasks, whereas GPT-3, a decoder model, leverages unidirectional context, massive model size, and the art of prompting to perform a wide range of tasks without the need for fine-tuning.

The choice between BERT and GPT-3 for a specific task depends on several factors, such as the availability of labeled data, computational resources, and the desired level of customization.

Computational Resource

Training and prompting a large model like GPT-3 requires powerful GPUs and a significant amount of memory. However, you don't need a powerful computer to use GPT-3. GPT-3 can be used directly

through OpenAI's API, making it more accessible to users without high-performance hardware through OpenAI's API, which handles the computational load for you. In contrast, BERT is much smaller, and can be trained, fine-tuned, and loaded on modern cunsumer-level hardware.

Monetary Cost

Using the OpenAI's API comes with its own monetary cost, as the usage of the API is billed based on the number of tokens processed, and GPT-3's large model size can result in higher costs for extensive usage. That said, the cost is considered low. For the 933 tweets considered in this tutorial, the cost for using ChatGPT (an extension of the GPT-3 model) with zero-shot prompting is about $0.39 and $0.65 with few-shot prompting (more below). On the other hand, fine-tuning BERT may require more time and effort to train but can run with low monetary cost if you have the necessary hardware.

Desired Level of Customization and the Need of Labeled Data

BERT is often more suitable for tasks that require a deeper understanding of context and benefit from fine-tuning on domain-specific labeled data.

On the other hand, GPT-3 is a powerful option for tasks that can leverage its vast knowledge and benefit from zero-shot or few-shot learning approaches without large amounts of labeled data. For example, when labeled data is scarce or the task is more general in nature. GPT-3 may perform well across various tasks without fine-tuning can be advantageous.

Open-source vs Closed-source

While BERT is an open-source model, GPT-3 is a closed-source model. This means that you can't access the source code of GPT-3, and you can't train your own GPT-3 model or modify the mode (and you also have to pay for the usage). However, not all LLMs are closed-source. For example, you can train your own GPT-2 model using the Hugging Face's implementation. In this tutorial, I will also show you another state-of-the-art open-source LLM, the FLAN-T5 model, which is suitable for stance detection tasks.

Now that we have a better understanding of the differences between BERT and GPT-3, let's dive deeper into prompting techniques for large language models in the next section.

9.2 Prompting Techniques for LLMs

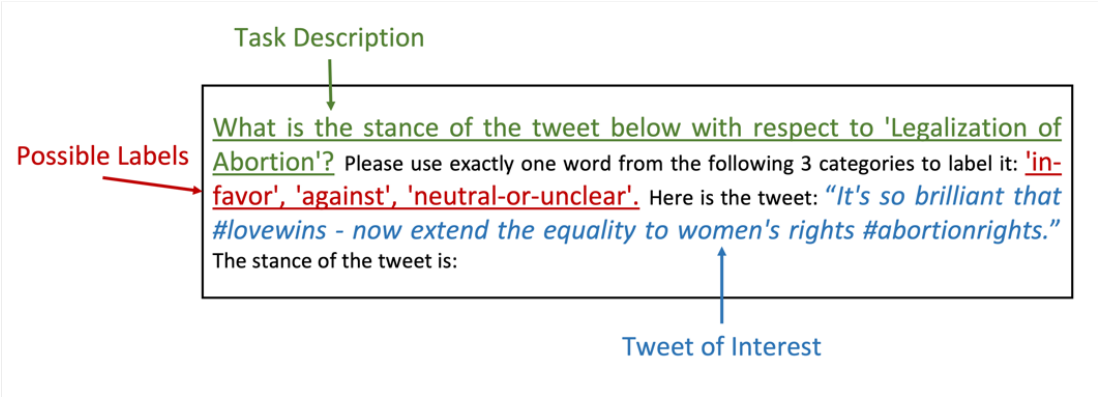
To effectively utilize LLMs like ChatGPT-3 for stance detection, we can employ various prompting techniques that guide the model's response without the need for fine-tuning.

> If you want to learn more about prompting techniques, I recommend this online tutorial: https://learnprompting.org/docs/intro

9.2.1 Zero-shot Prompting

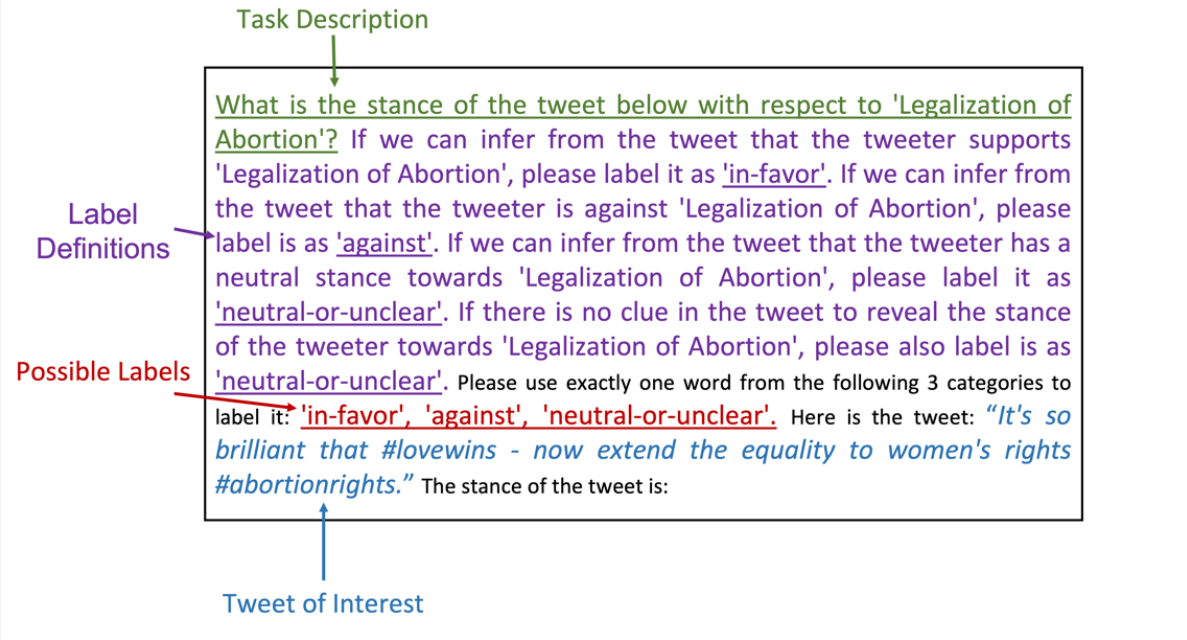
In the zero-shot prompting technique, for stance detection, GPT-3 is provided with a task description and with the tweet of interest, without any specific examples.

[6]:
```
display_resized_image_in_notebook("prompt_zero_shot_v1.png",0.3)
```



We can further improve this prompt with the definition of each stance category. Note that when human annotators are asked to label a tweet, they are usually provided with the definition of each stance category. To this end, I extract the definitions of each stance category from "codebook" of the SemEval 2016 Task 6 dataset and include them in the prompt.

[7]:
```
display_resized_image_in_notebook("prompt_zero_shot_v2.png",0.3)
```



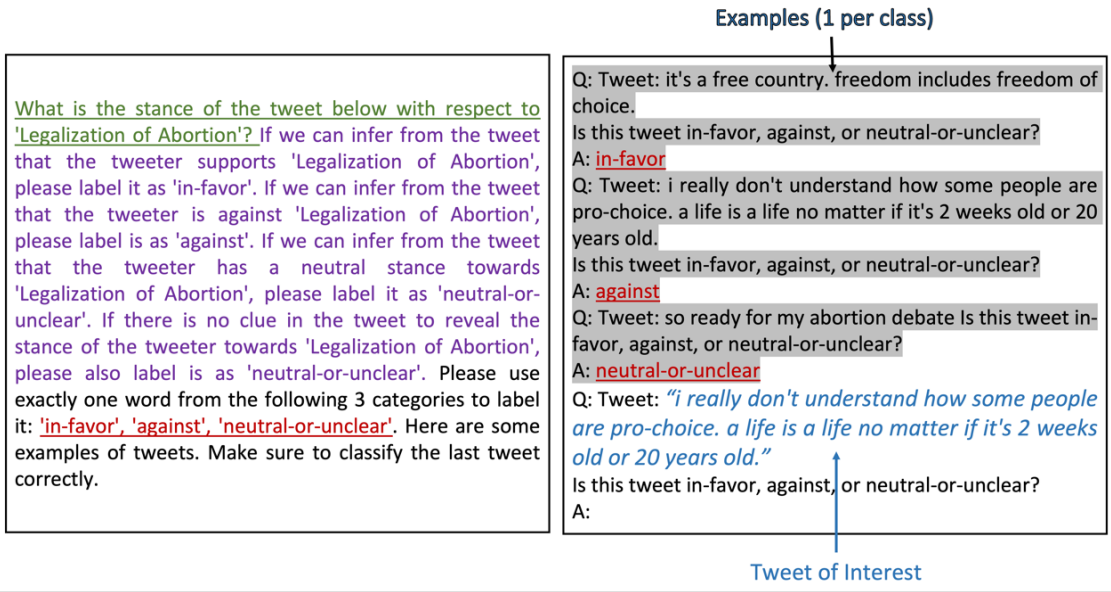
9.2.2 Few-shot Prompting

The few-shot prompting technique involves providing the LLM with a small number of examples to guide its response. This allows the model to learn from the provided examples and adapt its output accordingly. Below is an example of a few-shot prompt for stance detection, with 1 example tweet

for each stance category.

Note that I append the example tweets to the zero-shot prompt I made above.

[8]:
```
display_resized_image_in_notebook("prompt_few_shot.png",0.3)
```



Note: It may sound trivial, but when evaluating the performance of the LLMs with few-shot prompting, it is important to make sure that the examples provided to the model are not included in the test set to avoid data leakage.

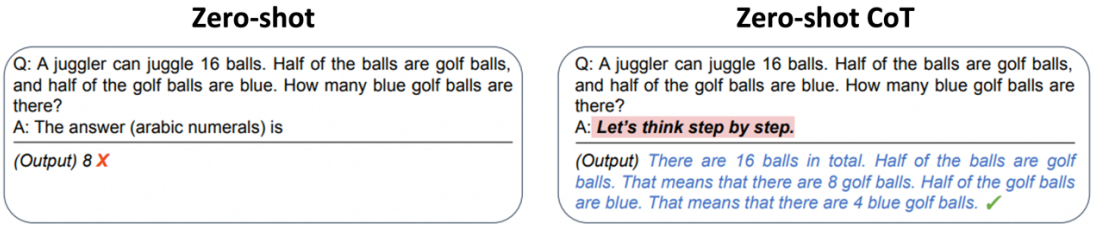
Note: A comprehensive evaluattion of the zero-shot and few-shot performance of GPT-3 is available in Brown, T. et al (2020).

9.2.3 Zero-shot Chain-of-thoughts (CoT) Prompting

By appending the words "let's think step-by-step" to the zero-shot prompt, we can ask GPT-3 to generate a chain of thoughts that lead to the stance prediction. This has been shown to improve the performance of LLMs in task that involves reasoning. This particular prompt is called zero-shot chain-of-thoughts (CoT) prompting.

One example task where the zero-shot chain-of-thoughts prompting technique has been shown to be effective is in task involves numerical reasoning. For instance, in the example below,, GPT-3 is able to generate a chain of thoughts that lead to the correct answer, as opposd to the standard zero-shot prompting technique, which fails to generate the correct answer.

[9]:
```
display_resized_image_in_notebook("prompt_cot_math.png",0.3)
```

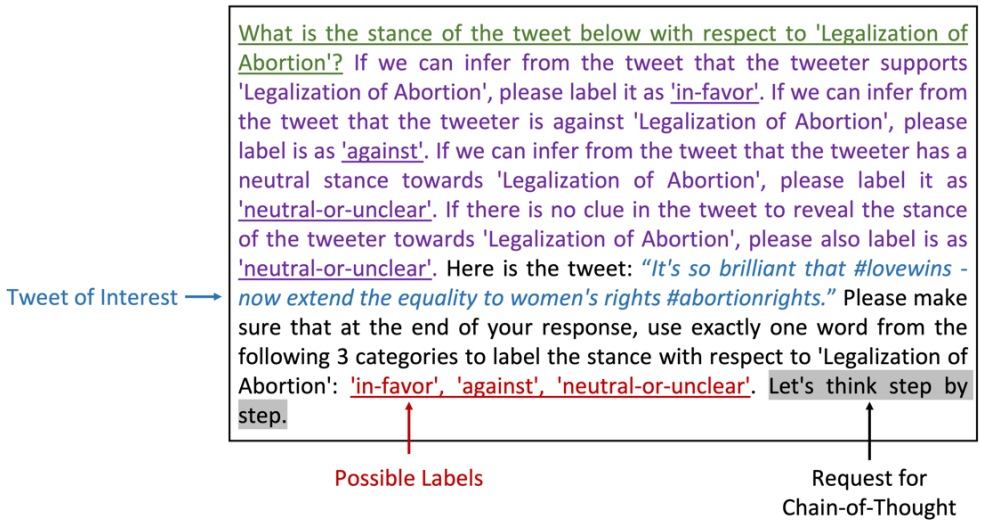


The zero-shot CoT is proposed by Kojima, T. et al (2022), and the figure is also modified from their paper.

Since stance detection often requires reasoning, it makes sense to consider zero-shot CoT prompting technique for our task.

Below is an example of a zero-shot chain-of-thoughts prompt for stance detection. Notice that I append the words "let's think step-by-step" to the zero-shot prompt I made above with minor modifications.

[10]:
```
display_resized_image_in_notebook("prompt_cot.png",0.3)
```



One important caveat about the chain-of-thought (CoT) prompt is that, according to Wei et al. (2022), CoT tends to yield performance gains only when used with larger models of around 100 billion parameters or more. For smaller language models, the CoT prompt might actually harm the performance, as the model may generate incorrect chains of thoughts, leading to an inaccurate prediction.

Note: The original chain-of-thoughts (CoT) prompt includes some examples of reasoning in the prompt, like the few-shot prompt. However, in this tutorial, for the sake of simplicity, I will only use the zero-shot CoT prompt. I will also use the term "CoT" to refer to the zero-shot CoT prompt.

9.3 Two state-of-the-art LLMs: ChatGPT and FLAN-T5

Before I move on to the programming part, I want to introduce two state-of-the-art LLMs that are suitable for stance detection tasks: ChatGPT and FLAN-T5.

9.3.1 ChatGPT

ChatGPT (`gpt-3.5-turbo`) is an extension of the GPT-3 model. One significant advantage of ChatGPT, compared to GPT-3 (`gpt-3-davinci`, the most powerful variant), is its lower cost. According to OpenAI's pricing page, using ChatGPT costs $0.002 per 1k tokens (approximately 750 English words, including both words in the prompt and generated sequence), which is about 10 times cheaper than GPT-3 at $0.02 per 1k tokens (as of 04/22/2023).

Besides the difference in pricing, the main distinction between ChatGPT and GPT-3 lies in their training approaches. While both models are pre-trained using the next word prediction task, ChatGPT is further trained with human feedback to generate more coherent and contextually relevant responses in conversational settings. This additional training is crucial since ChatGPT is designed to function as a chatbot, expected to provide responses that are coherent and contextually relevant to the user's input.

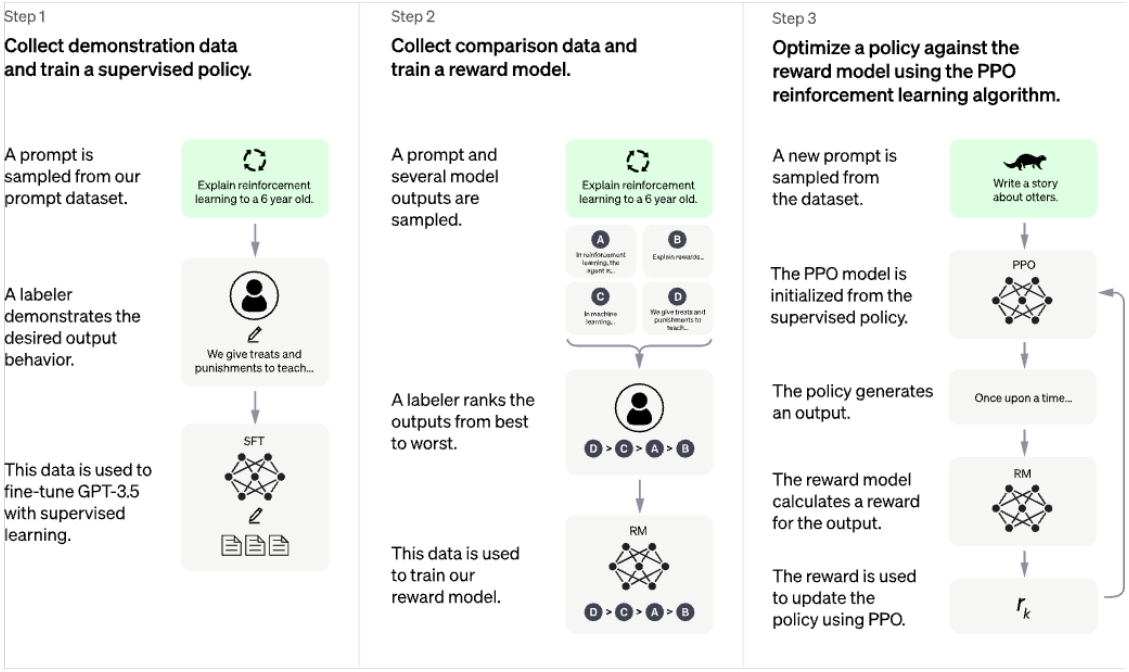
Finally, it is worth noting that there is a newer, more powerful variant of the model called GPT-4. This advanced version has demonstrated remarkable performance on various sophisticated exams, including AP exams, GRE tests, and the Law Bar Exam, among others. However, due to the higher cost and accessibility (as there is currently a waitlist to use it), this tutorial will focus on using `gpt-3.5-turbo`.

Fine-tuning with human feedback in ChatGPT During the fine-tuning of ChatGPT, a technique called **reinforcement learning from human feedback (RLHF)** is critical. The goal is to have ChatGPT generate texts that sound "human-like" in a conversation. This method includes making an initial dataset with the help of human AI trainers (who give conversations or answers to different prompts). Then, these human trainers compare and rank several responses created by Chat-GPT-3.5. Using these rankings, a reward model is trained to predict human's ranking. The ChatGPT model is then optimized to maximize the reward evluated by the reward model. By repeatedly using human feedback, the model gets better and its answers become more human-like.

The diagram below (copied from OpenAI's post on ChatGPT) shows the process of fine-tuning ChatGPT with RLHF.

> The details of the RLHF method is beyond the scope of this tutorial. If you are interested in learning more about RLHF, I recommend reading OpenAI's post on ChatGPT.

[11]:
```
display_resized_image_in_notebook("chatgpt_finetuning.png",scale = 1)
```



9.3.2 Open-source model FLAN-T5

The FLAN-T5 model is another LLM. A significant advantage of FLAN-T5 is that it is open-source, meaning that it is free to use (if you have access to GPUs)! This is in contrast to ChatGPT, which is a proprietary model.

It is based on the T5 (Text-to-Text Transfer Transformer) architecture developed by Google Research. Like GPT-3, FLAN-T5 also has the decoder component, which enables it to generate sequence given a prompt.

Instruction Fine-tuning One critical difference between FLAN-T5 and GPT-3 is the "instruction fine-tuning" procedure. After pre-training, FLAN-T5 is fine-tuned with over 1.8k text-to-text tasks, like summarization, translation, question-answering, among many, where data are fed into the model in `(input_text, output_text)` pairs to predict the output text given the input text.

This means that FLAN-T5 is more suitable for instruction-based question-answering tasks, such as stance detection, but is less suitable for long-form text generation tasks, such as story generation.

Below is a diagram of how FLAN-T5 is fine-tuned.

[12]:
```
display_resized_image_in_notebook("flan_t5_xxl.png",0.7)
```

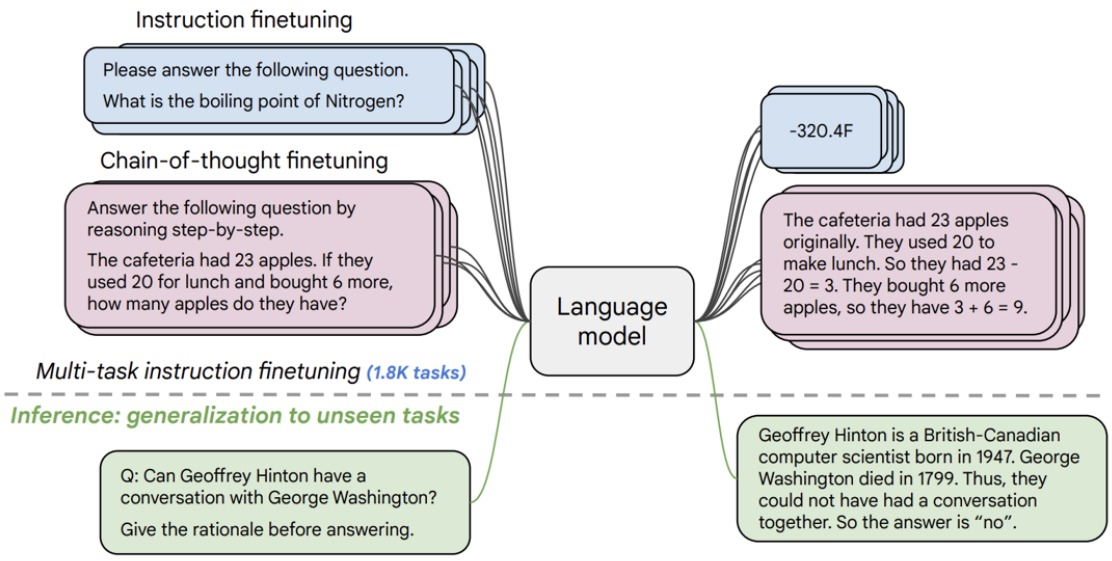


Image copied from: https://arxiv.org/pdf/2210.11416.pdf?trk=public_post_comment-text

Note for advanced readers: The intricate details of the FLAN-T5 model architecture and training procedure are beyond the scope of this tutorial. If you are interested in learning more about FLAN-T5, I recommend reading the blog post on T5. Although the blog post is about T5 model and not FLAN-T5 model, T5 model is the foundation of the FLAN-T5 model. Unlike GPT-3, a pure decoder model, FLAN-T5 is a "encoder-decoder model", which means it has both the encoder and the decoder. Because of the encoder component, FLAN-T5 is not pre-trained with the next word prediction task like GPT-3. Instead, FLAN-T5 is pre-trained using the "denoising autoencoder framework", which involves reconstructing the original input from a corrupted version of it. For curious readers who are eager to learn more, please check out the paper.

9.3.3 Critical Distinctions between ChatGPT and FLAN-T5

Closed-source vs. Open-source ChatGPT is a closed-source proprietary model developed by OpenAI, and using it comes with costs with API access. The implementation details and source code are not openly available, which limits users' ability to modify or understand the underlying workings of the model. On the other hand, FLAN-T5 is an open-source model, which means it is free to use, and the source code is publicly available for anyone to explore, use, and modify as needed.

Use case ChatGPT focuses on improving general conversational abilities and controllability and is fine-tuned with human feedback, while FLAN-T5 is designed to handle a wide range of short question-answering NLP tasks and is fine-tuned on text-to-text tasks. As a result, FLAN-T5 may be good at short question-answering task, but may not be as effective as ChatGPT for tasks that require longer responses, such as writing an essay.

Model size While OpenAI has not released the exact model size of ChatGPT, the GPT-3 model that is built upon has 175 billion parameters. On the other hand, the larges variant of FLAN-T5 (`flan-t5-xxl`) has about 11 billion parameters. The difference in model size entails that ChatGPT may capture more subtle meanings in language.

Model Variant	Number of Parameters
BERT-Large	340 million
FLAN-T5-Large	780 million
FLAN-T5-XXL	11 billion
ChatGPT-3.5	>175 billion (approx.)

> Note: FLAN-T5, like BERT, also comes in different variants, with different number of parameters. While FLAN-T5-XXL is the most powerful variant, it can not be run on Google Colab due to GPU memory limitations. In this tutorial, we will prompt the `flan-t5-large` variant. I have prompt the `flan-t5-xxl` variant elsewhere and upload the predictions so you can still view and evaluate the performance.

9.4 Programming Exercise: Implementing Stance Detection with FLAN-T5 and ChatGPT

Now that we have a basic understanding of ChatGPT and FLAN-T5, it's time to explore them through hands-on programming exercises in the context of stance detection on the Abortion dataset, which we used in the previous tutorial with BERT. These activities will allow you to gain some practical experience with both models, helping you to use them effectively for various NLP tasks while also revealing their differences in performance and implementation.

Don't worry if you're not an expert yet — let's just dive in and learn by doing as we implement stance detection using ChatGPT and FLAN-T5 and.

9.5 Read and Preprocess the Raw Data

Note that I am using the same preprocessed procedure as I did when I fine-tuned BERT in my previous tutorial, including removing retweet tags etc.

If you are interested in learning more about the preprocessing procedure, please refer to the previous tutorial.

[13]:
```
%load_ext autoreload
%autoreload 2
import pandas as pd
from os.path import join

from data_processor import SemEvalDataProcessor
```

```
from utils import convert_time_unit_into_name, get_parameters_for_dataset,␣
 ↪glob_re, list_full_paths, creat_dir_for_a_file_if_not_exists,␣
 ↪check_if_item_exist_in_nested_list,␣
 ↪get_dir_of_a_file,func_compute_metrics_sem_eval,partition_and_resample_df,␣
 ↪process_dataframe, tidy_name
```

[14]:
```
# set up
SEED = 42
TOPIC_OF_INTEREST = "Abortion"
DATASET = "SEM_EVAL"
par = get_parameters_for_dataset(DATASET)

PATH_OUTPUT_ROOT = join(par.PATH_RESULT_SEM_EVAL, "llm")
```

[15]:
```
# preprocess the data
sem_eval_data = SemEvalDataProcessor()
sem_eval_data.preprocess()

df_processed = sem_eval_data._read_preprocessed_data(topic=TOPIC_OF_INTEREST).
 ↪reset_index(drop=True)
# save the partitions (train, dev, test) for later use
df_partitions = sem_eval_data.partition_processed_data(seed=SEED,verbose=False)
```

[16]:
```
df_processed.head()
```

[16]:
```
     ID                                              tweet     topic    label  \
0  2312  i really don't understand how some people are ...  Abortion  AGAINST
1  2313  let's agree that it's not ok to kill a 7lbs ba...  Abortion  AGAINST
2  2314  @USERNAME i would like to see poll: how many a...  Abortion  AGAINST
3  2315  democrats are always against 'personhood' or w...  Abortion  AGAINST
4  2316  @USERNAME 'if you don't draw the line where i'...  Abortion     NONE

  partition
0     train
1     train
2     train
3     train
4     train
```

Let's look at the distribution of the stance labels across the training, testdation, and testing sets.

[17]:
```
# add a "count" column to count the number of tweets in each partition
df_label_dist = df_partitions[df_partitions.topic == TOPIC_OF_INTEREST].
 ↪value_counts(['partition','label']).sort_index()
df_label_dist
```

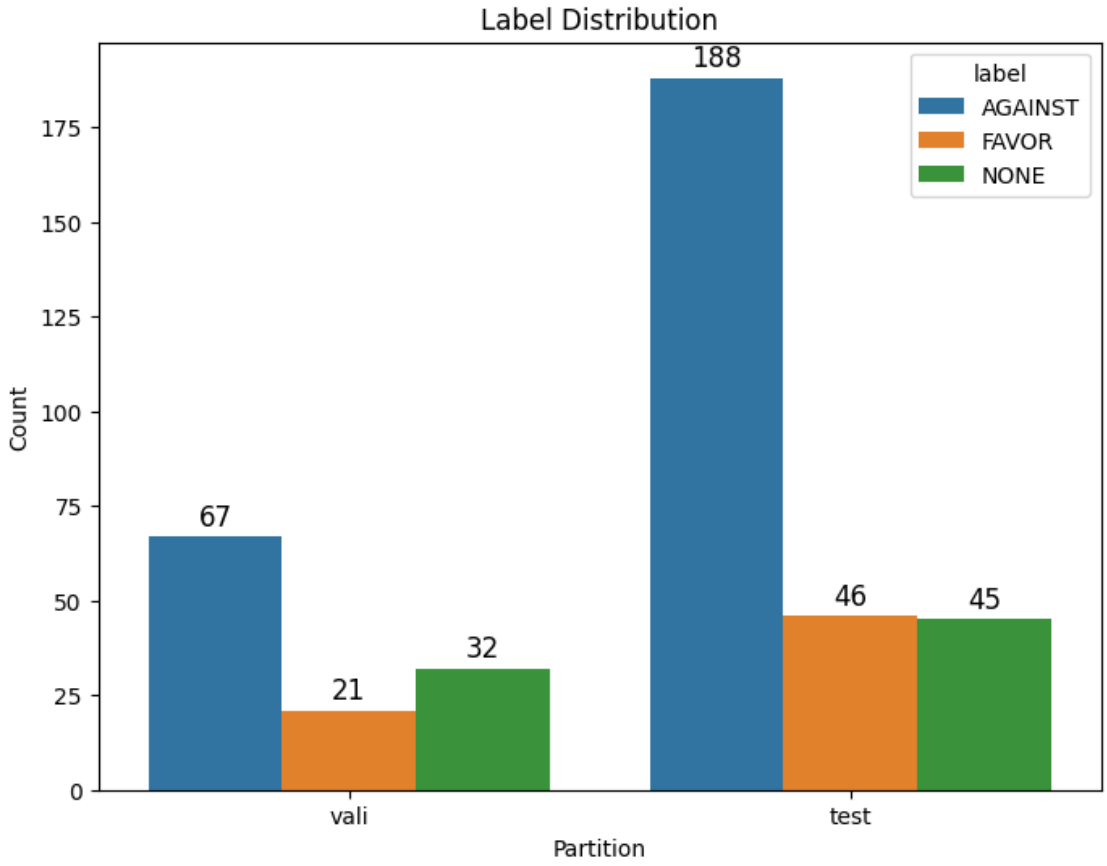
[17]:
```
partition  label
test       AGAINST    188
           FAVOR       46
           NONE        45
train      AGAINST    267
           FAVOR       83
           NONE       130
vali       AGAINST     67
           FAVOR       21
           NONE        32
dtype: int64
```

Critically, since we are using LLMs without fine-tuning, there is no need for a training set. We will only use the validation and testing sets for evaluation. The validation set can be employed to choose the right prompt and the right LLM, while the testing set is utilized to evaluate the final prompt and model.

From a practical standpoint, this approach significantly reduces the amount of labeled data required.

[18]:
```
import seaborn as sns
import matplotlib.pyplot as plt
# only keep the vali and test partitions
df_label_dist_plot = df_label_dist.reset_index()
df_label_dist_plot = df_label_dist_plot[df_label_dist_plot.partition.
 ↪isin(["vali", "test"])]


# Create the bar plot
plt.figure(figsize=(8, 6))
ax = sns.barplot(data=df_label_dist_plot, x="partition", y=0, hue="label",␣
 ↪order=["vali", "test"])

# Customize the plot
plt.xlabel("Partition")
plt.ylabel("Count")
plt.title("Label Distribution")

# Add count on top of each bar
for p in ax.patches:
    ax.annotate(
        f'{p.get_height():.0f}',
        (p.get_x() + p.get_width() / 2., p.get_height()),
        ha='center',
        va='baseline',
        fontsize=12,
        color='black',
        xytext=(0, 5),
        textcoords='offset points'
```

```
    )
# Show the plot
plt.show()
```



10 Create the Prompts

Let's create the prompts for the stance detection task. We will use the following three prompts: zero-shot, few-shot (1 example per class), and zero-shot chain-of-thought (CoT) prompts.

[19]:
```
%load_ext autoreload
%autoreload 2
from gpt_data_processor import SemEvalGPTDataProcessor
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

10.1 Zero-shot prompt

[20]:
```
# zero-shot (k = 0)
VERSION_PROMPT = "zero_shot"
```

[21]:
```
gpt_data_processor = SemEvalGPTDataProcessor(version_prompt = VERSION_PROMPT,␣
↪topic=TOPIC_OF_INTEREST)
# the prompt, which is the input to the LLMs
df_input_text = gpt_data_processor.embed_prompt(file_output=None,␣
↪write_csv=False,
                                                return_df=True)
```

[22]:
```
df_input_text.head()
```

[22]:

	ID	tweet	topic	label	\
0	2312	i really don't understand how some people are ...	Abortion	AGAINST	
1	2313	let's agree that it's not ok to kill a 7lbs ba...	Abortion	AGAINST	
2	2314	@USERNAME i would like to see poll: how many a...	Abortion	AGAINST	
3	2315	democrats are always against 'personhood' or w...	Abortion	AGAINST	
4	2316	@USERNAME 'if you don't draw the line where i'...	Abortion	NONE	

	partition	tweet_embedded
0	train	What is the stance of the tweet below with res...
1	train	What is the stance of the tweet below with res...
2	train	What is the stance of the tweet below with res...
3	train	What is the stance of the tweet below with res...
4	train	What is the stance of the tweet below with res...

Let's look at one example of the zero-shot prompt. As you can see, the tweet of interest is now embedded in the prompt.

[23]:
```
print("tweet:\n{}\n".format(df_input_text["tweet"][0]))
print("label:\n{}\n".format(df_input_text["label"][0]))
print("prompt:\n{}".format(df_input_text["tweet_embedded"][0]))
```

```
tweet:
i really don't understand how some people are pro-choice. a life is a life no
matter if it's 2 weeks old or 20 years old.

label:
AGAINST

prompt:
What is the stance of the tweet below with respect to 'Legalization of
Abortion'?  If we can infer from the tweet that the tweeter supports
'Legalization of Abortion', please label it as 'in-favor'. If we can infer from
the tweet that the tweeter is against 'Legalization of Abortion', please label
is as 'against'. If we can infer from the tweet that the tweeter has a neutral
```

```
stance towards 'Legalization of Abortion', please label it as 'neutral-or-
unclear'. If there is no clue in the tweet to reveal the stance of the tweeter
towards 'Legalization of Abortion', please also label is as 'neutral-or-
unclear'. Please use exactly one word from the following 3 categories to label
it: 'in-favor', 'against', 'neutral-or-unclear'. Here is the tweet. 'i really
don't understand how some people are pro-choice. a life is a life no matter if
it's 2 weeks old or 20 years old.' The stance of the tweet is:
```

10.2 Few-shot prompt

We repeat the same process for the few-shot prompt.

Note that the 3 examples are manually selected by me from the training set with the hope that they are representative of the stance class. These 3 examples are placed in the prompt for all the tweets while predicting the stance. There may be more effective ways to select the few-shot examples, but this is beyond the scope of this tutorial.

[24]:
```
gpt_data_processor = SemEvalGPTDataProcessor(version_prompt = "few_shot",␣
↪topic=TOPIC_OF_INTEREST)
# the prompt, which is the input to the LLMs
df_input_text = gpt_data_processor.embed_prompt(file_output=None,␣
↪write_csv=False,
                                                return_df=True)
```

[25]:
```
print("tweet:\n{}\n".format(df_input_text["tweet"][0]))
print("label:\n{}\n".format(df_input_text["label"][0]))
print("prompt:\n{}".format(df_input_text["tweet_embedded"][0]))
```

```
tweet:
i really don't understand how some people are pro-choice. a life is a life no
matter if it's 2 weeks old or 20 years old.

label:
AGAINST

prompt:
What is the stance of the tweet below with respect to 'Legalization of
Abortion'?  If we can infer from the tweet that the tweeter supports
'Legalization of Abortion', please label it as 'in-favor'. If we can infer from
the tweet that the tweeter is against 'Legalization of Abortion', please label
is as 'against'. If we can infer from the tweet that the tweeter has a neutral
stance towards 'Legalization of Abortion', please label it as 'neutral-or-
unclear'. If there is no clue in the tweet to reveal the stance of the tweeter
towards 'Legalization of Abortion', please also label is as 'neutral-or-
unclear'. Please use exactly one word from the following 3 categories to label
it: 'in-favor', 'against', 'neutral-or-unclear'.  Here are some examples of
tweets. Make sure to classify the last tweet correctly.
Q: Tweet: it's a free country. freedom includes freedom of choice.
Is this tweet in-favor, against, or neutral-or-unclear?
```

```
A: in-favor
Q: Tweet: i really don't understand how some people are pro-choice. a life is a
life no matter if it's 2 weeks old or 20 years old.
Is this tweet in-favor, against, or neutral-or-unclear?
A: against
Q: Tweet: so ready for my abortion debate
Is this tweet in-favor, against, or neutral-or-unclear?
A: neutral-or-unclear
Q: Tweet: 'i really don't understand how some people are pro-choice. a life is a
life no matter if it's 2 weeks old or 20 years old.'
Is this tweet in-favor, against, or neutral-or-unclear?
A:
```

10.3 Chain-of-thought prompt (CoT)

We repeat the same process for the zero-shot CoT prompt. The critical sentence "Let's think step by step." is added to the end of prompt.

[26]:
```
gpt_data_processor = SemEvalGPTDataProcessor(version_prompt = "CoT",␣
↪topic=TOPIC_OF_INTEREST)
# the prompt, which is the input to the LLMs
df_input_text = gpt_data_processor.embed_prompt(file_output=None,␣
↪write_csv=False,
                                                return_df=True)
```

[27]:
```
print("tweet:\n{}\n".format(df_input_text["tweet"][0]))
print("label:\n{}\n".format(df_input_text["label"][0]))
print("prompt:\n{}".format(df_input_text["tweet_embedded"][0]))
```

```
tweet:
i really don't understand how some people are pro-choice. a life is a life no
matter if it's 2 weeks old or 20 years old.

label:
AGAINST

prompt:
What is the stance of the tweet below with respect to 'Legalization of
Abortion'?  If we can infer from the tweet that the tweeter supports
'Legalization of Abortion', please label it as 'in-favor'. If we can infer from
the tweet that the tweeter is against 'Legalization of Abortion', please label
is as 'against'. If we can infer from the tweet that the tweeter has a neutral
stance towards 'Legalization of Abortion', please label it as 'neutral-or-
unclear'. If there is no clue in the tweet to reveal the stance of the tweeter
towards 'Legalization of Abortion', please also label is as 'neutral-or-
unclear'. Here is the tweet. 'i really don't understand how some people are pro-
choice. a life is a life no matter if it's 2 weeks old or 20 years old.' What is
the stance of the tweet with respect to 'Legalization of Abortion'? Please make
```

sure that at the end of your response, use exactly one word from the following 3 categories to label the stance with respect to 'Legalization of Abortion': 'in-favor', 'against', 'neutral-or-unclear'. Let's think step by step.

---

11 Feed the prompts to ChatGPT

Note that the specifc version of ChatGPT in used is `gpt-3.5-turbo`.

First, you have to decide whether you want to prompt ChatGPT on your own.

I recommend keeping `PROMPT_CHAT_GPT = False` (the default setting below) if you are running this notebook for the first time. This will read the predictions I made and uploaded to my GitHub repo, which will save you time and money.

If you want to try it out on your own, you can set `PROMPT_CHAT_GPT = True` and run the code below. In this case, you should also provide your own API key below. Here is the OpenAI's page on how to find your API key. Note that this will cost you about $1 to run this notebook.

[28]:
```
# Whether you want to prompt the GPT model yourself (this would cost around $1␣
↪USD for the entire tutorial)
PROMPT_CHAT_GPT = False
```

[29]:
```
import os
# You should set the API key for OpenAI
OPEN_AI_KEY = "FILL YOUR KEY HERE"

if PROMPT_CHAT_GPT:
    import openai
    openai.api_key = OPEN_AI_KEY
    assert OPEN_AI_KEY!="FILL YOUR KEY HERE", "You should set the API key for␣
↪OpenAI"

# to avoid overwriting the existing prediction file
for prompt_type in ["zero_shot", "few_shot", "CoT"]:
    file_prediction = join(PATH_OUTPUT_ROOT, "chatgpt_turbo_3_5", prompt_type,␣
↪"predictions.csv")
    file_prediction_cached = join(PATH_OUTPUT_ROOT,"chatgpt_turbo_3_5",␣
↪prompt_type, "predictions_cached.csv")
    # suffix the existing prediction file with "_cached"
    if PROMPT_CHAT_GPT:
        if os.path.exists(file_prediction):
            os.rename(file_prediction, file_prediction_cached)
    else:
        if os.path.exists(file_prediction_cached):
            os.rename(file_prediction_cached,file_prediction)
```

[30]:
```
from gpt_predict_label import GPTChatTurbo3_5LabelPredictor
```

[31]:
```
class PromptChatGPT_3_5:
    def __init__(self,prompt_version) -> None:
        # get the output paths
        # model_type_name = tidy_name(model_type)
        path_output = join(PATH_OUTPUT_ROOT, "chatgpt_turbo_3_5", prompt_version)
        self.file_output_predictions = join(path_output, "predictions.csv")

        gpt_data_processor =␣
↪SemEvalGPTDataProcessor(version_prompt=prompt_version, topic=TOPIC_OF_INTEREST)

        # create the prompt, which is the input to the LLMs
        df_input_text = gpt_data_processor.embed_prompt(file_output=None,␣
↪write_csv=False,
                                                        return_df=True)

        # partition the data into train, vali, test (note that we only need the␣
↪vali and test set in this approach)
        dict_df_single_domain = partition_and_resample_df(
            df_input_text, seed=None, partition_type="single_domain",
            read_partition_from_df=True,
            df_partitions=df_partitions)

        # select the partition to be labeled
        # - vali and test
        df_input_text_filtered = pd.DataFrame()
        for partition in ["vali_raw", "test_raw"]:
            df_input_text_filtered = pd.concat([df_input_text_filtered,␣
↪dict_df_single_domain[partition]])

        self.df_input_text = df_input_text_filtered

        # specify the output type (single-word or multi-word)
        if prompt_version in ["zero_shot","few_shot"]:
            mode_output = "single-word"
        elif prompt_version == "CoT":
            mode_output = "CoT"

        self.llm_label_predictor =␣
↪GPTChatTurbo3_5LabelPredictor(col_name_text="tweet_embedded",
                                                                ␣
↪col_name_label="stance_predicted",
                                                                 col_name_text_id=par.
↪TEXT_ID,
                                                                ␣
↪mode_output=mode_output)
    def estimate_cost(self):
```

```
        total_cost_estimate = self.llm_label_predictor.estimate_total_cost(self.
↪df_input_text)
        print("Estimated total cost: ${}".format(total_cost_estimate))
    def predict_labels(self):
        # prompt the LLM to make predictions (write the predictions to␣
↪`file_output_predictions`)
        self.llm_label_predictor.predict_labels(self.df_input_text,
                                                self.file_output_predictions,
                                                keep_tweet_id=True, keep_text=False,
                                                output_prob_mode=None,
                                                list_label_space=None,
                                                col_name_tweet_id=par.TEXT_ID)
    def load_predicted_labels(self):
        # load the predictions
        df_predictions = pd.read_csv(self.file_output_predictions)
        return df_predictions
```

[32]:
```
prompt_chat_gpt_zero_shot = PromptChatGPT_3_5("zero_shot")
```

[33]:
```
prompt_chat_gpt_zero_shot.estimate_cost()
```

```
Estimated total cost: $0.19196200000000002
```

[34]:
```
if PROMPT_CHAT_GPT:
  prompt_chat_gpt_zero_shot.predict_labels()
```

[35]:
```
prompt_chat_gpt_few_shot = PromptChatGPT_3_5("few_shot")
prompt_chat_gpt_few_shot.estimate_cost()
if PROMPT_CHAT_GPT:
  prompt_chat_gpt_few_shot.predict_labels()
```

```
Estimated total cost: $0.3164880000000003
```

[36]:
```
prompt_chat_gpt_cot = PromptChatGPT_3_5("CoT")
prompt_chat_gpt_cot.estimate_cost()
if PROMPT_CHAT_GPT:
  prompt_chat_gpt_cot.predict_labels()
```

```
Estimated total cost: $0.3324100000000002
```

11.1 View the predictions

[37]:
```
TEXT_ID_EXAMPLE = 9
```

[38]:
```
print("prompt:\n {}\n".format(prompt_chat_gpt_zero_shot.load_predicted_labels().
↪loc[TEXT_ID_EXAMPLE,"tweet_embedded"]))
```

```
print("GPT response:\n {}\n".format(prompt_chat_gpt_zero_shot.
 ↪load_predicted_labels().loc[TEXT_ID_EXAMPLE,"stance_predicted"]))
```

```
prompt:
 What is the stance of the tweet below with respect to 'Legalization of
Abortion'?  If we can infer from the tweet that the tweeter supports
'Legalization of Abortion', please label it as 'in-favor'. If we can infer from
the tweet that the tweeter is against 'Legalization of Abortion', please label
is as 'against'. If we can infer from the tweet that the tweeter has a neutral
stance towards 'Legalization of Abortion', please label it as 'neutral-or-
unclear'. If there is no clue in the tweet to reveal the stance of the tweeter
towards 'Legalization of Abortion', please also label is as 'neutral-or-
unclear'. Please use exactly one word from the following 3 categories to label
it: 'in-favor', 'against', 'neutral-or-unclear'. Here is the tweet. 'dear
religious right: i keep my uterus out of your church, so keep your church out my
uterus.' The stance of the tweet is:

GPT response:
 in-favor.
```

[39]:
```
print("prompt:\n {}\n".format(prompt_chat_gpt_few_shot.load_predicted_labels().
 ↪loc[TEXT_ID_EXAMPLE,"tweet_embedded"]))
print("GPT response:\n {}\n".format(prompt_chat_gpt_few_shot.
 ↪load_predicted_labels().loc[TEXT_ID_EXAMPLE,"stance_predicted"]))
```

```
prompt:
 What is the stance of the tweet below with respect to 'Legalization of
Abortion'?  If we can infer from the tweet that the tweeter supports
'Legalization of Abortion', please label it as 'in-favor'. If we can infer from
the tweet that the tweeter is against 'Legalization of Abortion', please label
is as 'against'. If we can infer from the tweet that the tweeter has a neutral
stance towards 'Legalization of Abortion', please label it as 'neutral-or-
unclear'. If there is no clue in the tweet to reveal the stance of the tweeter
towards 'Legalization of Abortion', please also label is as 'neutral-or-
unclear'. Please use exactly one word from the following 3 categories to label
it: 'in-favor', 'against', 'neutral-or-unclear'.  Here are some examples of
tweets. Make sure to classify the last tweet correctly.
Q: Tweet: it's a free country. freedom includes freedom of choice.
Is this tweet in-favor, against, or neutral-or-unclear?
A: in-favor
Q: Tweet: i really don't understand how some people are pro-choice. a life is a
life no matter if it's 2 weeks old or 20 years old.
Is this tweet in-favor, against, or neutral-or-unclear?
A: against
Q: Tweet: so ready for my abortion debate
Is this tweet in-favor, against, or neutral-or-unclear?
```

```
A: neutral-or-unclear
Q: Tweet: 'dear religious right: i keep my uterus out of your church, so keep
your church out my uterus.'
Is this tweet in-favor, against, or neutral-or-unclear?
A:

GPT response:
 in-favor
```

[40]:
```
print("prompt:\n {}\n".format(prompt_chat_gpt_cot.load_predicted_labels().
 ↪loc[TEXT_ID_EXAMPLE,"tweet_embedded"]))
print("GPT response:\n {}\n".format(prompt_chat_gpt_cot.load_predicted_labels().
 ↪loc[TEXT_ID_EXAMPLE,"stance_predicted"]))
```

```
prompt:
 What is the stance of the tweet below with respect to 'Legalization of
Abortion'?  If we can infer from the tweet that the tweeter supports
'Legalization of Abortion', please label it as 'in-favor'. If we can infer from
the tweet that the tweeter is against 'Legalization of Abortion', please label
is as 'against'. If we can infer from the tweet that the tweeter has a neutral
stance towards 'Legalization of Abortion', please label it as 'neutral-or-
unclear'. If there is no clue in the tweet to reveal the stance of the tweeter
towards 'Legalization of Abortion', please also label is as 'neutral-or-
unclear'. Here is the tweet. 'dear religious right: i keep my uterus out of your
church, so keep your church out my uterus.' What is the stance of the tweet with
respect to 'Legalization of Abortion'? Please make sure that at the end of your
response, use exactly one word from the following 3 categories to label the
stance with respect to 'Legalization of Abortion': 'in-favor', 'against',
'neutral-or-unclear'. Let's think step by step.

GPT response:
 The tweet implies that the tweeter supports the legalization of abortion and
believes that the religious right should not interfere with a woman's right to
choose. Therefore, the stance of the tweet with respect to 'Legalization of
Abortion' is 'in-favor'.
```

11.2 Evaluate the Predictions of Different Prompts

[41]:
```
from gpt_evaluate_labels import SemEvalGPTLabelEvaluator
from result_summarizer import ResultSummarizer
```

[42]:
```
def evaluate(model_type, prompt_version):
    gpt_data_processor = SemEvalGPTDataProcessor(topic=TOPIC_OF_INTEREST,
                                                 version_prompt=prompt_version)
    data_processor = SemEvalDataProcessor()
```

```
    # get the data with ground-truth labels and partition information (only␣
↪evaluate on the vali and test set)
    file_ground_truth = data_processor.
↪_get_file_processed_default(topic=TOPIC_OF_INTEREST)
    df = process_dataframe(
        input_csv=file_ground_truth,
        dataset=DATASET)
    df_partitions = data_processor.read_partitions(topic=TOPIC_OF_INTEREST)
    dict_df_single_domain = partition_and_resample_df(
        df, seed=None, partition_type="single_domain",
        read_partition_from_df=True,
        df_partitions=df_partitions)
    del dict_df_single_domain["train_raw"]


    # set the output path
    path_model_output =  join(PATH_OUTPUT_ROOT, model_type, prompt_version)
    file_output_metrics = join(path_model_output,"metrics.csv")
    file_output_confusion_mat = join(path_model_output,"confusion_matrix.csv")
    # also get the predictions
    file_input_predictions = join(path_model_output,"predictions.csv")

    # evaluate the predictions
    gpt_label_evaluator = SemEvalGPTLabelEvaluator(
        file_input_predictions=file_input_predictions,
        file_input_ground_truth=file_ground_truth,
        topic=TOPIC_OF_INTEREST, dataset=DATASET,
        model_gpt=model_type,
        num_examples_in_prompt=gpt_data_processor._get_num_examples_in_prompt(),
        key_join="ID",
        list_tweet_id_in_prompt=gpt_data_processor.
↪_get_list_tweet_id_in_prompt(),
        full_predictions=False)
    gpt_label_evaluator.evaluate(
        file_output_metrics=file_output_metrics,
        file_output_confusion_mat=file_output_confusion_mat,
        dict_df_eval=dict_df_single_domain,
        col_name_set="set")
```

[43]:
```
def summarize_results(list_model_type, list_prompt_version):
    df_hightlight_metrics = pd.DataFrame()
    for model_type in list_model_type:
        # summarize the results
        result_summarizer = ResultSummarizer(dataset=DATASET,
                                             ␣
↪list_version_output=list_prompt_version,
```

```
                                                eval_mode="single_domain",
                                                model_type="llm_" + model_type,
                                                task=None,
                                                file_name_metrics="metrics.csv",
                                               ␣
↪file_name_confusion_mat="confusion_matrix.csv",
                                               ␣
↪path_input_root=join(PATH_OUTPUT_ROOT, model_type),
                                               ␣
↪path_output=join(PATH_OUTPUT_ROOT,"summary"))
        # write the summary to a csv file
        df_hightlight_metrics_this = result_summarizer.
↪write_hightlight_metrics_to_summary_csv(
            list_metrics_highlight=['f1_macro', 'f1_NONE', 'f1_FAVOR',␣
↪'f1_AGAINST'],
            list_sets_highlight=['vali_raw', 'test_raw'],
            col_name_set="set")
        # visualize the confusion matrices and save the figures
        result_summarizer.visualize_confusion_metrices_over_domains_comb(
            ["vali_raw", "test_raw"],
            preserve_order_list_sets=True)

        df_hightlight_metrics = pd.concat([df_hightlight_metrics,␣
↪df_hightlight_metrics_this], axis=0)
    return df_hightlight_metrics
```

[44]:
```
evaluate("chatgpt_turbo_3_5", "zero_shot")
evaluate("chatgpt_turbo_3_5", "few_shot")
evaluate("chatgpt_turbo_3_5", "CoT")
```

```
/content/prelim_stance_detection/scripts/utils.py:713: FutureWarning:
load_metric is deprecated and will be removed in the next major version of
datasets. Use 'evaluate.load' instead, from the new library Hugging Face Evaluate:
https://huggingface.co/docs/evaluate
  metric_computer[name_metric] = load_metric(name_metric)
Downloading builder script:   0%|          | 0.00/2.32k [00:00<?, ?B/s]
Downloading builder script:   0%|          | 0.00/1.65k [00:00<?, ?B/s]
Downloading builder script:   0%|          | 0.00/2.52k [00:00<?, ?B/s]
Downloading builder script:   0%|          | 0.00/2.58k [00:00<?, ?B/s]
```

[45]:
```
df_hightlight_metrics_chatgpt = summarize_results(["chatgpt_turbo_3_5"],␣
↪["zero_shot","few_shot","CoT"])

# reorder the rows
```

```
df_hightlight_metrics_chatgpt['model_type'] = pd.
 ↪Categorical(df_hightlight_metrics_chatgpt['model_type'],␣
 ↪categories=["llm_chatgpt_turbo_3_5"], ordered=True)
df_hightlight_metrics_chatgpt['version'] = pd.
 ↪Categorical(df_hightlight_metrics_chatgpt['version'],␣
 ↪categories=["zero_shot","few_shot","CoT"], ordered=True)

# rename columns
df_hightlight_metrics_chatgpt =␣
 ↪df_hightlight_metrics_chatgpt[["model_type","version","set","f1_macro","f1_NONE","f1_FAVOR","
 ↪sort_values(by=["model_type","version"]).rename(columns={"version":
 ↪"prompt_type","set":"partition"})
```

```
<Figure size 1500x500 with 0 Axes>
```

```
<Figure size 1500x500 with 0 Axes>
```

```
<Figure size 1500x500 with 0 Axes>
```

[46]:
```
df_hightlight_metrics_chatgpt
```

[46]:

	model_type	prompt_type	partition	f1_macro	f1_NONE	f1_FAVOR	f1_AGAINST
1	llm_chatgpt_turbo_3_5	zero_shot	vali_raw	0.631255	0.652632	0.666667	0.574468
2	llm_chatgpt_turbo_3_5	zero_shot	test_raw	0.507138	0.435644	0.672269	0.413502
4	llm_chatgpt_turbo_3_5	few_shot	vali_raw	0.777948	0.800000	0.740741	0.793103
5	llm_chatgpt_turbo_3_5	few_shot	test_raw	0.637211	0.563380	0.676923	0.671329
7	llm_chatgpt_turbo_3_5	CoT	vali_raw	0.450072	0.512397	0.512821	0.325000
8	llm_chatgpt_turbo_3_5	CoT	test_raw	0.387721	0.360000	0.568421	0.234742

11.2.1 View the performance on the validation set

[47]:
```
df_hightlight_metrics_chatgpt_vali =␣
 ↪df_hightlight_metrics_chatgpt[df_hightlight_metrics_chatgpt["partition"].
 ↪isin(["vali_raw"])]

# convert "vali_raw" to "vali" (in the partition column)
df_hightlight_metrics_chatgpt_vali.
 ↪loc[df_hightlight_metrics_chatgpt_vali["partition"]=="vali_raw","partition"] =␣
 ↪"vali"
```

```
df_hightlight_metrics_chatgpt_vali
```

[47]:
```
                model_type prompt_type partition  f1_macro   f1_NONE  f1_FAVOR  \
1  llm_chatgpt_turbo_3_5   zero_shot      vali  0.631255  0.652632  0.666667
4  llm_chatgpt_turbo_3_5    few_shot      vali  0.777948  0.800000  0.740741
7  llm_chatgpt_turbo_3_5         CoT      vali  0.450072  0.512397  0.512821

   f1_AGAINST
1    0.574468
4    0.793103
7    0.325000
```

The first two columns indicate the model type and the prompt type, respectively. The third column indicates that the performance is evaluated on the validation set. The `f1_macro` column indicates the macro-averaged F1 score (across the three stance types). We use this value to quantify the overall performance of the model. Note that we are using the `f1_macro` metric rather than accuracy because the dataset is imbalanced, and the `f1_macro` metric is more robust to imbalanced datasets.

> To learn more about macro-F1 score, I recommend taking a look at this tutorial https://towardsdatascience.com/micro-macro-weighted-averages-of-f1-score-clearly-explained-b603420b292f#:~:text=The%20macro%2Daveraged%20F1%20score,regardless%20of%20their%20s

Based on the macro-F1 scores, the best performing combination is ChatGPT using the few-shot prompt. On the other hand, the zero-shot CoT prompt appears to negatively impact performance. In the next section, we will delve into this further by examining the confusion matrix to better understand these results.

The last 3 columns indicate the performance of each stance type. For few-shot prompt, this shows that the model is better at predicting the `AGAINST` and `FAVOR` stance than the `NONE` stance.

> Note: In practice, to avoid data leakage, when selecting the best combination of prompt type and model type, we should use the performance on the validation set to choose the best combination, and then use the performance on the test set to evaluate the final model.

11.2.2 View the performance on the test set

[48]:
```
df_hightlight_metrics_chatgpt_test =␣
↪df_hightlight_metrics_chatgpt[df_hightlight_metrics_chatgpt["partition"].
↪isin(["test_raw"])]

# convert "test_raw" to "test" (in the partition column)
df_hightlight_metrics_chatgpt_test.
↪loc[df_hightlight_metrics_chatgpt_test["partition"]=="test_raw","partition"] =␣
↪"test"

df_hightlight_metrics_chatgpt_test
```

[48]:
```
              model_type prompt_type partition  f1_macro   f1_NONE  f1_FAVOR  \
2  llm_chatgpt_turbo_3_5   zero_shot      test  0.507138  0.435644  0.672269
5  llm_chatgpt_turbo_3_5    few_shot      test  0.637211  0.563380  0.676923
8  llm_chatgpt_turbo_3_5         CoT      test  0.387721  0.360000  0.568421

   f1_AGAINST
2    0.413502
5    0.671329
8    0.234742
```

The results are similar to the validation set. The best prompt type is `ChatGPT-turbo-3.5` with the `few-shot` prompt. The zero-shot CoT prompt only seems to hurt the performance.

11.2.3 Compare ChatGPT with BERT on the test set

Let's compare the performance of FLAN-T5 with BERT. The results of BERT are generated from the previous tutorial.

[49]:
```
df_hightlight_metrics_bert = pd.read_csv(join(par.
 ↪PATH_RESULT_SEM_EVAL_TUNING,"summary","metrics_highlights.csv"))
df_hightlight_metrics_bert =␣
 ↪df_hightlight_metrics_bert[["version","set","f1_macro","f1_NONE","f1_FAVOR","f1_AGAINST"]][df
 ↪set.isin(["test_raw"])].rename(columns={"version":"model_type"})
# reorder the rows
df_hightlight_metrics_bert['model_type'] = pd.
 ↪Categorical(df_hightlight_metrics_bert['model_type'],␣
 ↪categories=["bert-base-uncased","vinai_bertweet_base","kornosk_polibertweet_mlm"],␣
 ↪ordered=True)
df_hightlight_metrics_bert = df_hightlight_metrics_bert.
 ↪sort_values(by=["model_type"]).reset_index(drop=True)
# rename columns
df_hightlight_metrics_bert = df_hightlight_metrics_bert.rename(columns={"set":
 ↪"partition"})
# convert "test_raw" to "test" (in the partition column)
df_hightlight_metrics_bert.
 ↪loc[df_hightlight_metrics_bert["partition"]=="test_raw","partition"] = "test"

df_hightlight_metrics_bert
```

[49]:
```
                 model_type partition  f1_macro  f1_NONE  f1_FAVOR  f1_AGAINST
0        bert-base-uncased      test    0.4748   0.4196    0.4275      0.5775
1      vinai_bertweet_base      test    0.5797   0.5323    0.5401      0.6667
2  kornosk_polibertweet_mlm      test    0.5616   0.5440    0.4762      0.6645
```

Based on the macro-F1 scores, the ChatGPT using the zero-shot prompt outperforms all the BERT variants we examined in the previous tutorial.

11.3 Analyzing the Confusion Matrix for Deeper Insights

Interestingly, the few-shot prompt emerges as the most effective prompt type for ChatGPT, while the CoT prompt performs poorly. To gain a deeper understanding of this phenomenon, let's examine the confusion matrix.

> Note: Examining the confusion matrix is essential because it provides a detailed overview of the model's performance across different classes. It reveals not only the correct predictions (true positives) but also the instances where the model made errors (false positives and false negatives). By analyzing the confusion matrix, we can identify patterns in misclassifications and gain insights into the strengths and weaknesses of the model. Here is a great tutorial on how to interpret the confusion matrix and its relationships with macro-F1 scores: https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826

[50]:
```
from IPython.display import Image, display
```

Below are the confusion matrices for the few-shot prompt.

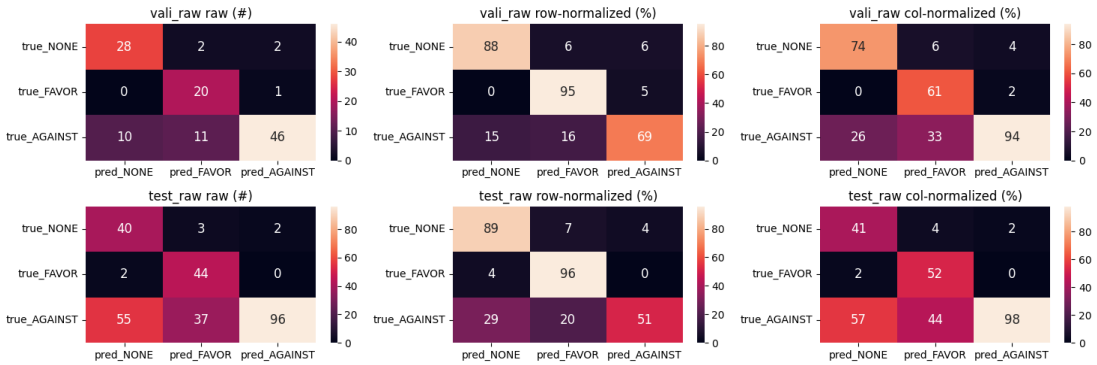
The first row are the matrices for the validation set, and the second row are the matrices for the test set.

Each row of matrices consists of three types:

1. The leftmost matrices are the raw confusion matrices.
2. The middle matrices show the confusion matrices normalized by row (i.e., the sum of each row equals 100). In these matrices, the diagonal values correspond to the recall value of each class.
3. The rightmost matrices illustrate the confusion matrices normalized by column (i.e., the sum of each column equals 100). In these matrices, the diagonal values are the precision value for each class.

In each matrix, the rows represent the true labels, and the columns represent the predicted labels. The diagonal elements denote correct predictions, while the off-diagonal elements indicate incorrect predictions.

[51]:
```
display_resized_image_in_notebook(join(PATH_OUTPUT_ROOT,"summary","chatgpt_turbo_3_5_few_shot_c
↪png"))
```

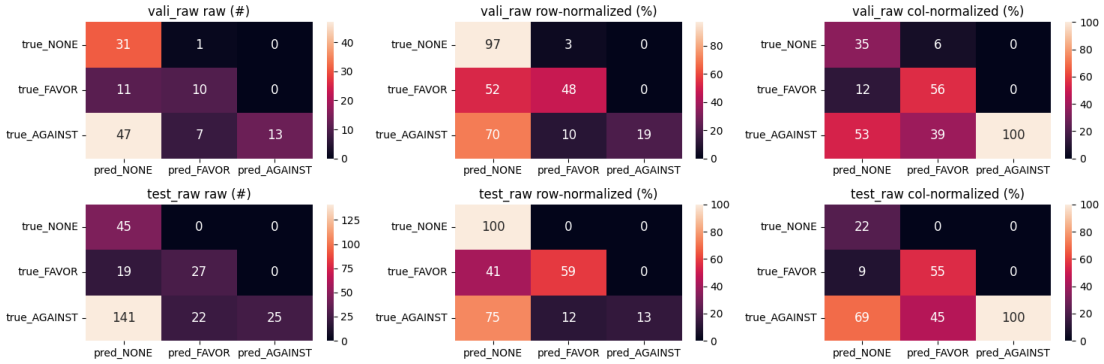


As shown in the test set confusion matrices, the model demonstrates proficiency in distinguishing between the `FAVOR` and `NONE` stances. However, it faces challenges in accurately predicting the `AGAINST` stance, frequently misclassifying them as `NONE`.

Now, let's examine the confusion matrix for the CoT prompt, which has the worst macro-F1 score.

The confusion matrices below reveal that the model has a strong tendency to predict the `NONE` stance, which is defined as the stance being neutral or unclear. It seems like after reasoning step by step, the model tends to reach a conclusion that the stance is not clear.

[52]:
```
display_resized_image_in_notebook(join(PATH_OUTPUT_ROOT,"summary","chatgpt_turbo_3_5_CoT_comb_c
↪png"))
```



---

12 Feed the prompts to FLAN-T5 models

Great! Now we know that ChatGPT with few-shot prompts outperforms fine-tuned BERT on this dataset. This is promising because prompting the LLM doesn't require a large amount of labeled data for training. However, one notable downside is that using ChatGPT incurs a monetary cost.

Next, let's explore if the open-source FLAN-T5 can achieve similar results.

First, you have to decide whether you want to prompt FLAN-T5 on your own.

I recommend keeping `PROMPT_FLAN_T5 = False` (the default setting below) if you are running this notebook for the first time. This will read the predictions I made and uploaded to my GitHub repo, which will save you time.

If you want to try it out on your own, you can set `PROMPT_CHAT_GPT = True` and run the code below. If you are running this on Google Colab, make sure you have use the GPU run time. To do this, go to `Runtime` -> `Change runtime type` -> `Hardware accelerator` -> `GPU`. This will ensure that the note will run more efficiently and quickly.

[53]:
```
PROMPT_FLAN_T5 = False
```

[54]:
```
%load_ext autoreload
%autoreload 2
from gpt_predict_label import FlanT5LargeLabelPredictor, FlanT5XxlLabelPredictor
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

12.1 Helpfer function

[55]:
```
def prompt_flan_t5(model_type, prompt_version):
    # get the output paths
    # model_type_name = tidy_name(model_type)
    path_output = join(PATH_OUTPUT_ROOT, model_type, prompt_version)
    file_output_predictions = join(path_output, "predictions.csv")

    gpt_data_processor = SemEvalGPTDataProcessor(version_prompt=prompt_version,␣
↪topic=TOPIC_OF_INTEREST)

    # create the prompt, which is the input to the LLMs
    df_input_text = gpt_data_processor.embed_prompt(file_output=None,␣
↪write_csv=False,
                                                    return_df=True)

    # partition the data into train, vali, test (note that we only need the vali␣
↪and test set in this approach)
    dict_df_single_domain = partition_and_resample_df(
        df_input_text, seed=None, partition_type="single_domain",
        read_partition_from_df=True,
        df_partitions=df_partitions)

    # select the partition to be labeled
    # - vali and test
    df_input_text_filtered = pd.DataFrame()
    for partition in ["vali_raw", "test_raw"]:
        df_input_text_filtered = pd.concat([df_input_text_filtered,␣
↪dict_df_single_domain[partition]])

    df_input_text = df_input_text_filtered

    # specify the output type (single-word or multi-word)
    if prompt_version in ["zero_shot","few_shot"]:
        mode_output = "single-word"
    elif prompt_version == "CoT":
        mode_output = "CoT"

    if model_type == "flan-t5-large":
```

```
        llm_label_predictor =␣
↪FlanT5LargeLabelPredictor(col_name_text="tweet_embedded",
                                                          ␣
↪col_name_label="stance_predicted",
                                                           col_name_text_id=par.
↪TEXT_ID,
                                                           try_use_gpu=True,
                                                          ␣
↪per_device_eval_batch_size=16,
                                                           mode_output=mode_output,
                                                           for_generation_only=True)
    elif model_type == "flan-t5-xxl":
        llm_label_predictor =␣
↪FlanT5XxlLabelPredictor(col_name_text="tweet_embedded",
                                                          ␣
↪col_name_label="stance_predicted",
                                                           col_name_text_id=par.
↪TEXT_ID,
                                                           try_use_gpu=True,
                                                          ␣
↪per_device_eval_batch_size=16,
                                                           mode_output=mode_output,
                                                           for_generation_only=True)
    else:
        raise ValueError("model_type not supported")

    # prompt the LLM to make predictions (write the predictions to␣
↪`file_output_predictions`)
    llm_label_predictor.predict_labels(df_input_text,
                                       file_output_predictions,
                                       keep_tweet_id=True, keep_text=False,
                                       output_prob_mode=None,
                                       list_label_space=None,
                                       col_name_tweet_id=par.TEXT_ID)
```

12.2 Use the zero-shot and few-shot prompts

Note that as mentioned earlier, zero-shot CoT prompt only works when the LLM has more than 100 billion parameters. Even the larges variant of FLAN-T5 (`flan-t5-xxl`) has only 11 billion parameters, so we will not use the zero-shot CoT prompt in this tutorial.

[56]:
```
model_type = "flan-t5-large"
if PROMPT_FLAN_T5:
    for prompt_version in ["zero_shot","few_shot"]:
        print("model:{}, prompt version: {}".format(model_type, prompt_version))
        prompt_flan_t5(model_type, prompt_version)
```

While FLAN-T5-XXL is the most powerful variant, it can not be run on Google Colab due to GPU memory limitations. In this tutorial, I have run the predictions elsewhere and upload the predictions. If you want to prompt FLAN-T5-XXL on your own and you have access to a large GPU with more memory (>30GB), you can prompt the FLAN-T5-XXL model by setting `LARGE_GPU_AVAILABLE = True` below.

[57]:
```
# run this only if use you a large enough GPU (>30GB)
LARGE_GPU_AVAILABLE = False
if LARGE_GPU_AVAILABLE and PROMPT_FLAN_T5:
    model_type = "flan-t5-xxl"
    for prompt_version in ["zero_shot","few_shot"]:
        print("model:{}, prompt version: {}".format(model_type, prompt_version))
        prompt_flan_t5(model_type, prompt_version)
```

12.2.1 View the predictions

[58]:
```
def read_predictions(model_type, prompt_version):
    # get the output paths
    # model_type_name = tidy_name(model_type)
    path_output = join(PATH_OUTPUT_ROOT, model_type, prompt_version)
    file_output_predictions = join(path_output, "predictions.csv")

    # read the predictions
    df_predictions = pd.read_csv(file_output_predictions)
    return df_predictions
```

As demonstrated below, using a zero-shot prompt, the FLAN-T5-XXL model successfully predicts the stance of this example tweet, whereas the FLAN-T5-Large model struggles to do so.

[59]:
```
df_predictions_flan_xxl_zero_shot = read_predictions("flan-t5-xxl","zero_shot")
df_predictions_flan_large_zero_shot =␣
↪read_predictions("flan-t5-large","zero_shot")
```

[60]:
```
# view the first prediction
print("prompt:\n{}\n".
↪format(df_predictions_flan_xxl_zero_shot["tweet_embedded"][0]))
# print("ID:\n{}\n".format(df_predictions_flan_xxl_zero_shot["ID"][0]))
print("true label::\n{}\n".
↪format(df_input_text[df_input_text["ID"]=="2313"][["label"]].values[0][0].
↪lower()))

print("FLAN-T5-LARGE's prediction:\n{}".
↪format(df_predictions_flan_large_zero_shot["stance_predicted"][0]))
print("FLAN-T5-XXL's prediction:\n{}".
↪format(df_predictions_flan_xxl_zero_shot["stance_predicted"][0]))
```

```
prompt:
What is the stance of the tweet below with respect to 'Legalization of
```

```
Abortion'?  If we can infer from the tweet that the tweeter supports
'Legalization of Abortion', please label it as 'in-favor'. If we can infer from
the tweet that the tweeter is against 'Legalization of Abortion', please label
is as 'against'. If we can infer from the tweet that the tweeter has a neutral
stance towards 'Legalization of Abortion', please label it as 'neutral-or-
unclear'. If there is no clue in the tweet to reveal the stance of the tweeter
towards 'Legalization of Abortion', please also label is as 'neutral-or-
unclear'. Please use exactly one word from the following 3 categories to label
it: 'in-favor', 'against', 'neutral-or-unclear'. Here is the tweet. 'let's agree
that it's not ok to kill a 7lbs baby in the uterus @USERNAME #dnc #clinton2016
@USERNAME #procompromise' The stance of the tweet is:

true label::
against

FLAN-T5-LARGE's prediction:
in-favor
FLAN-T5-XXL's prediction:
against
```

12.3 Evaluate the Predictions of Different Prompts and Different Models

Fantastic! With the predictions in hand from two types of prompts and two variants of FLAN-T5 models, it's time to evaluate their performance across all tweets in both the validation and test sets.

[61]:
```
evaluate("flan-t5-large", "zero_shot")
evaluate("flan-t5-large", "few_shot")
# although you may not have run the xxl model due to GPU contraint, you can␣
↪still evaluate the results because I have already run the model elsewhere and␣
↪saved the predictions
evaluate("flan-t5-xxl", "zero_shot")
evaluate("flan-t5-xxl", "few_shot")
```

[62]:
```
# sumamrize the result acrpss different models and prompt versions
# - also visualize the confusion matrices
df_hightlight_metrics_flan_t5 =␣
↪summarize_results(["flan-t5-large","flan-t5-xxl"],["zero_shot","few_shot"])

df_hightlight_metrics_flan_t5 =␣
↪df_hightlight_metrics_flan_t5[df_hightlight_metrics_flan_t5.model_type.
↪isin(["llm_flan-t5-large","llm_flan-t5-xxl"])]

# reorder the rows
df_hightlight_metrics_flan_t5['model_type'] = pd.
↪Categorical(df_hightlight_metrics_flan_t5['model_type'],␣
↪categories=["llm_flan-t5-large","llm_flan-t5-xxl"], ordered=True)
```

```
df_hightlight_metrics_flan_t5['version'] = pd.
 ↪Categorical(df_hightlight_metrics_flan_t5['version'],␣
 ↪categories=["zero_shot","few_shot"], ordered=True)
df_hightlight_metrics_flan_t5 = df_hightlight_metrics_flan_t5.
 ↪sort_values(by=["model_type","version"]).reset_index(drop=True)

# rename columns
df_hightlight_metrics_flan_t5 =␣
 ↪df_hightlight_metrics_flan_t5[["model_type","version","set","f1_macro","f1_NONE","f1_FAVOR","
 ↪sort_values(by=["model_type","version"]).rename(columns={"version":
 ↪"prompt_type","set":"partition"})
```

```
<Figure size 1500x500 with 0 Axes>
```

```
<Figure size 1500x500 with 0 Axes>
```

```
<Figure size 1500x500 with 0 Axes>
```

```
<Figure size 1500x500 with 0 Axes>
```

12.3.1 View the performance on the validation set

[63]:
```
df_hightlight_metrics_flan_t5_vali =␣
 ↪df_hightlight_metrics_flan_t5[df_hightlight_metrics_flan_t5["partition"].
 ↪isin(["vali_raw"])]

# convert "vali_raw" to "vali" (in the partition column)
df_hightlight_metrics_flan_t5_vali.
 ↪loc[df_hightlight_metrics_flan_t5_vali["partition"]=="vali_raw","partition"] =␣
 ↪"vali"

df_hightlight_metrics_flan_t5_vali
```

[63]:

	model_type	prompt_type	partition	f1_macro	f1_NONE	f1_FAVOR	f1_AGAINST
0	llm_flan-t5-large	zero_shot	vali	0.244478	0.060606	0.215686	0.457143
2	llm_flan-t5-large	few_shot	vali	0.267255	0.060606	0.192771	0.548387
4	llm_flan-t5-xxl	zero_shot	vali	0.693311	0.600000	0.653846	0.826087
6	llm_flan-t5-xxl	few_shot	vali	0.680935	0.653846	0.603774	0.785185

The first two columns indicate the model type and the prompt type, respectively. The third column indicates that the performance is evaluated on the validation set. The `f1_macro` column indicates the macro-averaged F1 score (across the three stance types). We use this value to quantify the overall

performance of the model. Note that we are using the `f1_macro` metric rather than accuracy because the dataset is imbalanced, and the `f1_macro` metric is more robust to imbalanced datasets.

Based on the macro-F1 scores, the larger FLAN-T5-XXL model performs better than the smaller FLAN-T5-Large model across two prompt types. This is expected because the larger model has more parameters and can capture more subtle meanings in language.

For FLAN-T5-XXL model, using few-shot prompt does not seem to help.

The last 3 columns indicate the performance of each stance type. This shows that the model is better at predicting the `AGAINST` stance than the `NONE` and `FAVOR` stances.

> Note that, in practice, to avoid data leakage, when selecting the best combination of prompt type and model type, we should use the performance on the validation set to choose the best combination, and then use the performance on the test set to evaluate the final model.
>
> To learn more about macro-F1 score, I recommend taking a look at this tutorial https://towardsdatascience.com/micro-macro-weighted-averages-of-f1-score-clearly-explained-b603420b292f#:~:text=The%20macro%2Daveraged%20F1%20score,regardless%20of%20their%20s

12.3.2 View the performance on the test set

[64]:
```
df_hightlight_metrics_flan_t5_test =␣
↪df_hightlight_metrics_flan_t5[df_hightlight_metrics_flan_t5["partition"].
↪isin(["test_raw"])]

# convert "test_raw" to "test" (in the partition column)
df_hightlight_metrics_flan_t5_test.
↪loc[df_hightlight_metrics_flan_t5_test["partition"]=="test_raw","partition"] =␣
↪"test"

df_hightlight_metrics_flan_t5_test
```

[64]:

	model_type	prompt_type	partition	f1_macro	f1_NONE	f1_FAVOR	f1_AGAINST
1	llm_flan-t5-large	zero_shot	test	0.212933	0.043478	0.181034	0.414286
3	llm_flan-t5-large	few_shot	test	0.278003	0.083333	0.198953	0.551724
5	llm_flan-t5-xxl	zero_shot	test	0.619033	0.583333	0.531250	0.742515
7	llm_flan-t5-xxl	few_shot	test	0.591850	0.534653	0.519685	0.721212

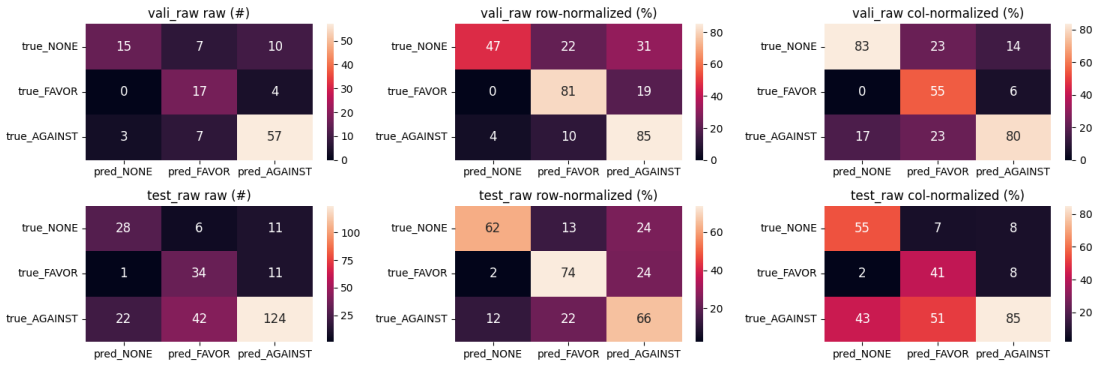
The results are similar to the validation set. The larger FLAN-T5-XXL model performs better than the smaller FLAN-T5-Large model across two prompt types. For FLAN-T5-XXL model, using few-shot prompt does not seem to help.

12.3.3 Confusion Matrix

As we did for ChatGPT, let's examine the confusion matrix for best performing combination of model and prompt (i.e., FLAN-T5-XXL with zero-shot prompt).

Similar to ChatGPT with few-shot prompt (the best combination for ChatGPT), the FLAN-T5-XXL model also has trouble classifying the tweets with the `AGAINST` stance, misclassifying them as either `FAVOR` or `NONE`. In addition, the model also misclassifies some `FAVOR` and `NONE` tweets as `AGAINST`. These collectivly result in a lower macro-F1 score than ChatGPT with few-shot prompt.

[65]:
```
display_resized_image_in_notebook(join(PATH_OUTPUT_ROOT,"summary","flan-t5-xxl_zero_shot_comb_c
↪png"))
```



12.3.4 Compare FLAN-T5 with BERT

Let's compare the performance of FLAN-T5 with BERT. The results of BERT are generated from the previous tutorial.

[66]:
```
df_hightlight_metrics_bert = pd.read_csv(join(par.
↪PATH_RESULT_SEM_EVAL_TUNING,"summary","metrics_highlights.csv"))
df_hightlight_metrics_bert =␣
↪df_hightlight_metrics_bert[["version","set","f1_macro","f1_NONE","f1_FAVOR","f1_AGAINST"]].
↪rename(columns={"version":"model_type"})
# reorder the rows
df_hightlight_metrics_bert['model_type'] = pd.
↪Categorical(df_hightlight_metrics_bert['model_type'],␣
↪categories=["bert-base-uncased","vinai_bertweet_base","kornosk_polibertweet_mlm"],␣
↪ordered=True)
df_hightlight_metrics_bert = df_hightlight_metrics_bert.
↪sort_values(by=["model_type"]).reset_index(drop=True)
# rename columns
df_hightlight_metrics_bert = df_hightlight_metrics_bert.rename(columns={"set":
↪"partition"})
# convert "test_raw" to "test" (in the partition column)
```

```
df_hightlight_metrics_bert.
 ↪loc[df_hightlight_metrics_bert["partition"]=="test_raw","partition"] = "test"
# subset the test set
df_hightlight_metrics_bert_test =␣
 ↪df_hightlight_metrics_bert[df_hightlight_metrics_bert["partition"]== "test"]
df_hightlight_metrics_bert_test
```

[66]:

	model_type	partition	f1_macro	f1_NONE	f1_FAVOR	f1_AGAINST
2	bert-base-uncased	test	0.4748	0.4196	0.4275	0.5775
5	vinai_bertweet_base	test	0.5797	0.5323	0.5401	0.6667
8	kornosk_polibertweet_mlm	test	0.5616	0.5440	0.4762	0.6645

Based on the macro-F1 scores, the FLAN-T5-XXL model using the zero-shot prompt outperforms all the BERT variants we examined in the previous tutorial.

---

13 Compare ChatGPT with FLAN-T5 and BERT on the test set

[67]:
```
df_hightlight_metrics_llm_test = \
  pd.concat([df_hightlight_metrics_flan_t5_test ,␣
 ↪df_hightlight_metrics_chatgpt_test],
           axis=0)

df_hightlight_metrics_llm_test.sort_values(by=["f1_macro"],ascending=False).
 ↪reset_index(drop=True)
```

[67]:

	model_type	prompt_type	partition	f1_macro	f1_NONE	f1_FAVOR	f1_AGAINST
0	llm_chatgpt_turbo_3_5	few_shot	test	0.637211	0.563380	0.676923	0.671329
1	llm_flan-t5-xxl	zero_shot	test	0.619033	0.583333	0.531250	0.742515
2	llm_flan-t5-xxl	few_shot	test	0.591850	0.534653	0.519685	0.721212
3	llm_chatgpt_turbo_3_5	zero_shot	test	0.507138	0.435644	0.672269	0.413502
4	llm_chatgpt_turbo_3_5	CoT	test	0.387721	0.360000	0.568421	0.234742
5	llm_flan-t5-large	few_shot	test	0.278003	0.083333	0.198953	0.551724
6	llm_flan-t5-large	zero_shot	test	0.212933	0.043478	0.181034	0.414286

[68]:
```
# bert
df_hightlight_metrics_bert_test.sort_values(by=["f1_macro"],ascending=False).
↪reset_index(drop=True)
```

[68]:

	model_type	partition	f1_macro	f1_NONE	f1_FAVOR	f1_AGAINST
0	vinai_bertweet_base	test	0.5797	0.5323	0.5401	0.6667
1	kornosk_polibertweet_mlm	test	0.5616	0.5440	0.4762	0.6645
2	bert-base-uncased	test	0.4748	0.4196	0.4275	0.5775

The results of the comparison between ChatGPT, FLAN-T5, and BERT models across different prompts show that the prompting approach can outperform a fine-tuned BERT, even when considering domain-specific pretrained models like kornosk_polibertweet_mlm and vinai_bertweet_base.

ChatGPT with few-shot prompts achieves the highest macro-F1 scores on the test set, followed by FLAN-T5-XXL with zero-shot and few-shot prompts. Notably, even the open-source large language model, FLAN-T5, reaches decent performance, offering a cost-effective alternative to ChatGPT. However, it's important to note that using FLAN-T5 requires access to GPUs with about 30GB of memory to handle its large size.

These large language models with prompting strategies deliver better performance than the fine-tuned BERT variants, including the domain-specific models. This demonstrates the potential of using prompts with large language models, as they offer competitive performance without requiring extensive labeled data for training.

However, it is important to note that fine-tuning BERT has a distinct advantage in that its performance can be straightforwardly improved by collecting more labeled data. The performance of a fine-tuned BERT model increases with the amount of labeled training data available. This is not the case for large language models using prompting strategies, as the number of examples that can be fit in a prompt is limited.

> Note for advanced readers: To ensure that the prompting approach benefits from a larger amount of training data, you can also fine-tune an LLM on a substantial amount of labeled data. For more information, refer to the bonus section below.

---

14 Conclusions

In conclusion, this tutorial has demonstrated how to implement stance detection using ChatGPT and FLAN-T5 on the Abortion dataset. We've explored the use of different prompt types and compared their performance to BERT-based models, including domain-specific pre-trained models. The results show that prompting large language models, like ChatGPT and FLAN-T5, can outperform fine-tuned BERT models in this task, even without extensive labeled data for training. However, it's essential to consider the trade-offs, such as the monetary cost of using ChatGPT and the memory requirements for utilizing FLAN-T5.

14.1 Bonus for eager readers

In addition to using prompting strategies with large language models, it's worth noting that fine-tuning LLMs is another viable approach to improve their performance on specific tasks, including stance detection. By fine-tuning an LLM on a task-specific dataset, the model can adapt to the nuances of the data, better understand the domain-specific language, and potentially yield higher performance.

OpenAI has an guide on how to fine-tune GPT-3 on a specific task. It is also possible to fine-tune open-source LLMs like FLAN-T5 for specific tasks. For example, this tutorial demonstrates how to fine-tune FLAN-T5 for text classification. One caveat is that fine-tuning LLMs, like fine-tuning a BERT model, also requires a large amount of labeled data for training.

Another caveat is that fine-tining the GPT-3 model and using a fine-tuned model can be expensive. Please see the OpenAI's pricing page for more details. On the other hand, fine-tuning FLAN-T5-XXL is GPU intensive and requires about 680GB of GPU memory and few days of training time.