Fine-Tuning Text Transformers with Gene Ontology Annotations: A Case Study

This repository serves as an exercise to demonstrate the use of definitions of Gene Ontology (GO) terms to fine-tune a pre-trained BERT-based Large Language Model (LLM). The goal is to extract hidden dependencies among manually annotated GO term definitions, leveraging the model to categorize these definitions based on their alignment with the major GO ontologies: *Biological Process*, *Cellular Component*, and *Molecular Function*.

> **Fine-tuning** is a powerful technique in machine learning that involves using models that have undergone **semi-supervised training** on extensive data from various sources. This process imparts the model with underlying semantic features of the language. After pre-training, the model is further trained in a **supervised manner** for a specific task, such as text generation, classification, or label prediction (for instance, identifying whether emails are spam or not). During this phase, the deeper encoded features of the pre-trained model are retained, but the trainable parameters of the output layer(s) are adjusted. For example, a feed-forward neural network followed by a softmax function can be added for binary classification of "spam" vs "not spam." These steps embody the concept of Transfer Learning, a powerful machine learning technique that involves applying knowledge learned from one task effectively to another task.

This post is also available for reading on Medium, where you can find additional insights and details.

For further information about BERT, please refer to the official documentation.

Clone Repository

```
(base)$ git clone https://github.com/matiollipt/GO-graph-definition-text-transformer.git
```

Configure Conda Environment

- Creating the environment

```
(base)$: conda env create -f environment.yml
```

- Activating the environment

```
(base)$: conda activate go_tune
```

- Running the notebook

```
(go_tune)$: python -m jupyter main.ipynb
```

Introduction

Transformers have gained significant attention in the Natural Language Processing (NLP) and machine learning landscape, contributing to the development of generative models like chatGPT. Text transformer-based architectures are specifically engineered to **analyze the dependencies between words (or *tokens*) in a text sequence, considering their positions to accurately capture the meaning of the text**. This capability is instrumental in tasks such as text classification and ranking, as well as enabling generative models to generate new sentences in response to prompts.

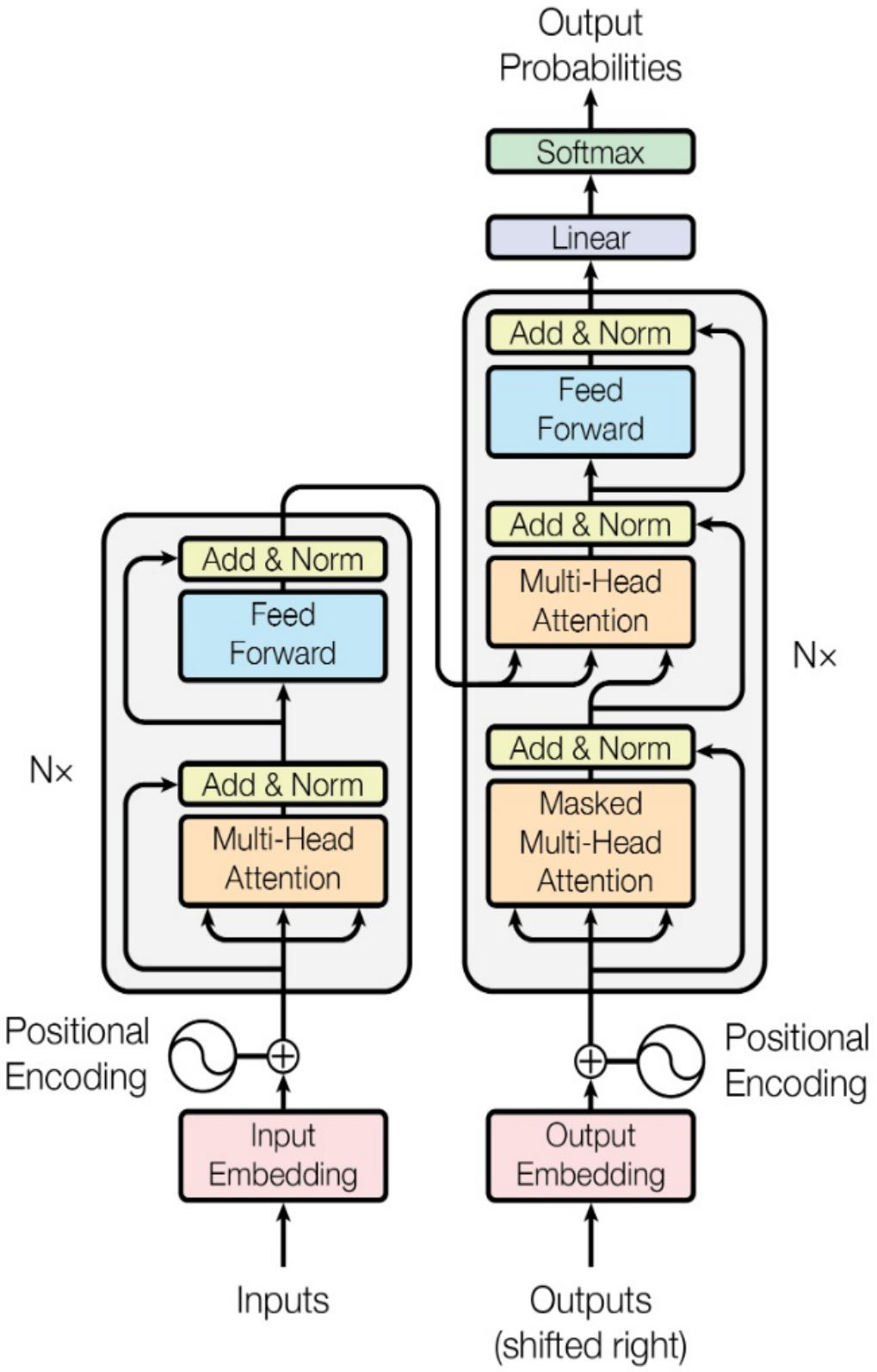


Figure 1: The Transformer - model architecture.

The transformer architecture can be adapted and trained for more specific applications, such as ProteinBERT. ProteinBERT builds upon the efficient transformer architecture and is trained on over 106 million protein sequences from UniProt to capture protein semantics. The pre-trained model can then be fine-tuned to classify proteins into families, identify phylogenetic relationships, predict protein function and subcellular localization, and anticipate protein-protein interactions, among other interesting applications for protein embeddings.

Automating Discovery In Life Sciences

High-throughput DNA sequencing methods) has give us the power to sequence the entire genome of a species within a day.

With the sequence on hands, we can deploy **computational models to identify and annotate genes** based on their sequences, attributing correct functions and subcellular locations .

For example, the Critical Assessment of Protein Function Annotation (CAFA)00166-2) competition engages the data science community in enhancing protein prediction by utilizing features derived from Gene Ontology (GO).

The Gene Ontology Graph

The Gene Ontology is represented as a **directed acyclic graph (DAG**) where **each node represents a specific GO term**. Each GO term defines a particular aspect of genes and their products.

The GO terms are organized into **three main aspects**:

- ***Molecular Function (MF)***: These terms define the **activities** performed by gene products, such as *catalysis* or *transport*. These functions can be further refined by more specific GO terms, for example, "protein kinase activity" within the broader category of "catalysis".
- ***Cellular Component (CC)***: These terms specify the subcellular **locations** of gene products, including compartments like *chloroplast* or *nucleus*, as well as macromolecular complexes like *proteasome* or *ribosome*.
- ***Biological Process (BP)***: These terms delineate the biological **pathways** in which gene products are involved, ranging from *DNA repair* and *carbohydrate metabolic process* to overarching processes like *biosynthetic processes*.

The relationships between these terms are **hierarchical**, with **parent-child relationships** indicating broader and more specific terms, respectively. This hierarchical structure allows researchers to **annotate genes and gene products**, providing valuable information about their functions and roles in biological processes. For more information about how the GO graph is structured, please refer to my repository GO-graph-EDA and the Gene Ontology reference. For now, it is essential to know that **each node representing a GO term has specific attributes**.

Extracting GO terms definitions

A feature often overlooked when deploying the GO graph to assist in the prediction of gene function is the **textual definition of each GO term**. For example, the GO term ID GO:0015986 is defined as *"The transport of protons across the plasma membrane to generate an electrochemical gradient (proton-motive force) that powers ATP synthesis."*, along with other attributes shown below:

GO id: GO:0015986

- **name**: 'proton motive force-driven plasma membrane ATP synthesis'
- **namespace**: 'biological_process'
- **def**: '"The transport of protons across the plasma membrane to generate an electrochemical gradient (proton-motive force) that powers ATP synthesis." [GOC:mtg_sensu, ISBN:0716731363]'
- **synonym**: ['"ATP synthesis coupled proton transport" BROAD []', '"plasma membrane ATP synthesis coupled proton transport" EXACT []']
- **is_a**: ['GO:0015986']}

The nodes contain attributes describing the corresponding GO terms, and these are the essential ones that appear in every node:

- ***name***: unique identifier of the term in a human-readable format
- ***namespace***: one of the three major ontologies (MF, CC or BP) to which the term belongs
- ***definition***: a short description of what the GO term means for humans. It can also contains references to publications defining the term (e.g. PMID:10873824).

Dataset Preparation

Our initial step involves pre-processing the dataset to make it compatible with the pre-trained model that we are going to fine-tune according to our needs. The BERT model has previously undergone **semi-supervised training** on an extensive dataset comprising millions of entries from Wikipedia in 102 different languages. During this pre-training phase, the model learns language rules and dependencies, setting the stage for its application in various **supervised training** scenarios. These

applications include tasks like sentiment analysis, text generation, text sequence classification, and, in our specific case, identifying the major GO ontology categories (BP, CC, and MF) for the given text sequences.

To prepare the dataset for fine-tuning our model, we will perform the following tasks:

1. **Feature Extraction**: In this step, we convert the attributes associated with each GO term's nodes into a Pandas DataFrame. This conversion streamlines the creation of pre-processed and tokenized datasets that will be used to train the model. We will focus on extracting only the text definitions and labels corresponding to the aspects we aim to predict (BP, MF, and CC).
2. **Dataset Creation and Splitting**: Our data will be divided into two subsets: a training set (80%) and an test set (20%). These sets will contain the input text derived from the definitions of GO terms.
3. **Text Tokenization**: The input text undergoes a tokenization process, breaking it into smaller units known as tokens. Special tokens are also incorporated to indicate the start and end of sequences and sentences. This step is crucial for enabling the model to comprehend the text.

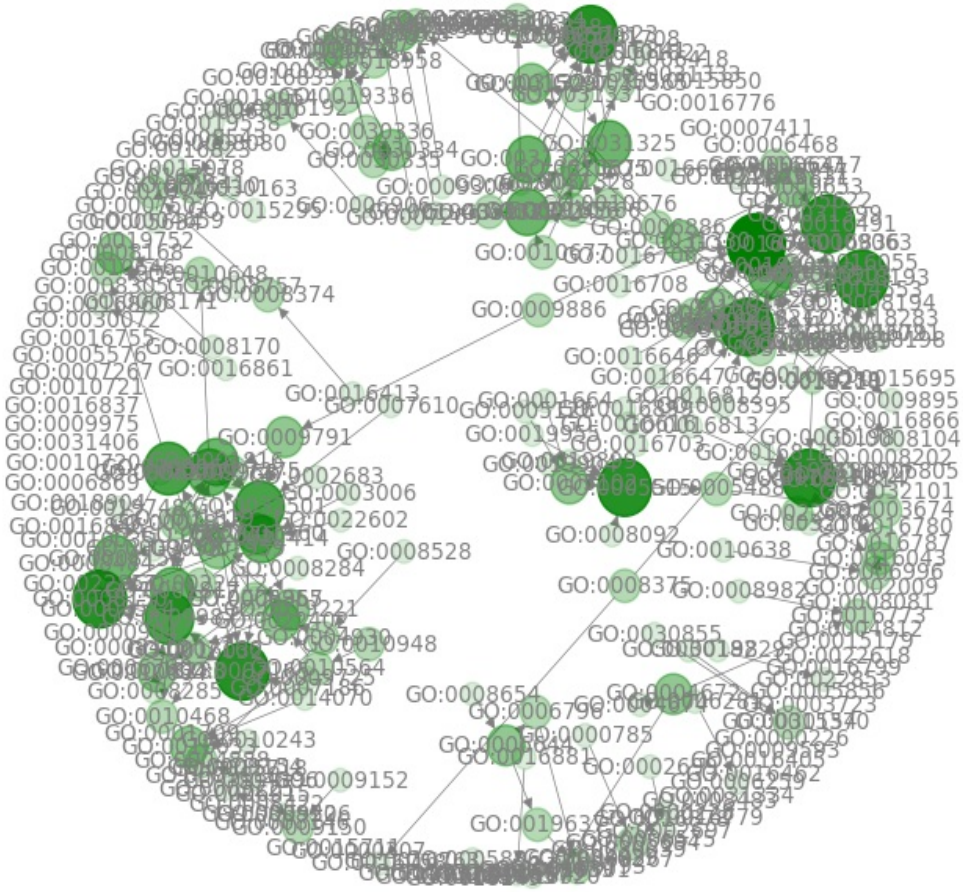
To create and manage these datasets, we will make use of the Dataset library provided by Hugging Face. Additionally, as our project progresses, we will leverage pre-trained sequence classification models and their corresponding tokenizers, which are also available from Hugging Face.

Feature Extraction

The GO graph is stored in the *OBO (Open Biomedical Ontologies) file format*, designed specifically for the construction and representation of biological ontologies. To convert this file format into a **NetworkX** object, we can utilize the Python library obonet. NetworkX offers a robust framework for graph manipulation and analysis.

```
from obonet import read_obo

go_graph = read_obo(home_dir.joinpath("data/go-basic.obo"))
```

We can use the custom function `plot_graph( )` available in the notebook to visualize a subset of GO graph nodes and how they are connected. Here, I selected the nodes with the **highest degrees** to plot. For further details about nodes' degrees, please refer to my GitHub repository GO-graph-EDA.



The next step involves **extracting information from nodes in the GO graph and structuring it into a dataframe** to streamline the handling of text definitions. We also obtain the number of words to choose the maximum length for input during training and evaluation. Shorter inputs reduce the training time and computational requirements a great deal.

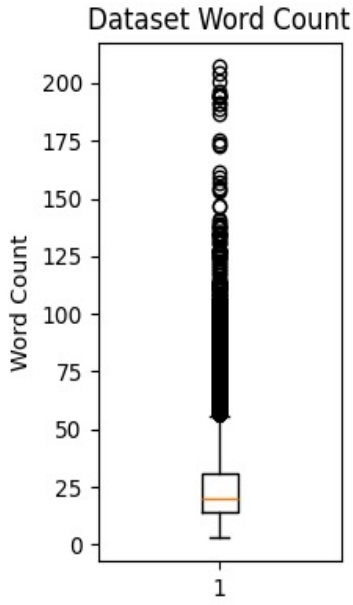
During this extraction process, we intend to focus on the textual description from the definitions, which is stored in a specific format. Each node in the GO graph has an ID and a dictionary containing various attributes, including the definition. For example:

```
('GO:0000001', {'name': 'mitochondrion inheritance', 'namespace': 'biological_process', 'def': '"The dis
```

There are many ways to create dataframes from other types of unstructured or structured data. The code snippet below does the job and saves the dataframe:

```
# create GO definitions dataframe

# create empty dataframe to store nodes' attributes
go_df = pd.DataFrame(
    columns=["go_id", "name", "aspect", "definition", "def_word_count"]
)

# iterate over nodes to extract dictionary keys and values
n_rows = len(go_graph.nodes)

for idx, item in tqdm(enumerate(go_graph.nodes.items()), total=n_rows):
    go_term = item[0]
    name = item[1]["name"]
    aspect = item[1]["namespace"]

    # split 'def' content and get the text definition only
```

```
    definition = item[1]["def"].split(sep='"', maxsplit=2)[1]

    # count the number of words of the text we just extracted
    def_word_count = len(re.findall(r"\w+", definition))

    go_df.loc[idx] = [
        go_term,
        name,
        aspect,
        definition,
        def_word_count,
    ]

# save dataframe
home_dir.joinpath("data/").mkdir(parents=True, exist_ok=True)
go_df.to_csv(home_dir.joinpath("data/go_df.csv"), index=False)

# word count boxplot
plt.figure(figsize=(2, 4))
plt.title("Dataset Word Count")
plt.boxplot(go_df.def_word_count.values)
plt.ylabel("Word Count")
plt.show()
```



Before proceeding, we will make some modifications in our dataset: renaming the **definition** column to *text*, and converting the **aspect** column to a **categorical data type** while mapping the aspects to numeric labels and keeping only these two columns for the downstream tasks:

```
# loading saved dataframe with GO graph nodes' attributes

# load saved GO dataframe
go_df = pd.read_csv(home_dir.joinpath("data/go_df.csv"))

# convert categorical labels to numbers (aspect)
go_df["aspect"] = pd.Categorical(go_df["aspect"])

# get categorical codes
go_df["label"] = go_df["aspect"].cat.codes

# select only relevant columns
data = go_df[["definition", "label"]].copy()

# rename definition column
data.rename(columns={"definition": "text"}, inplace=True)
```

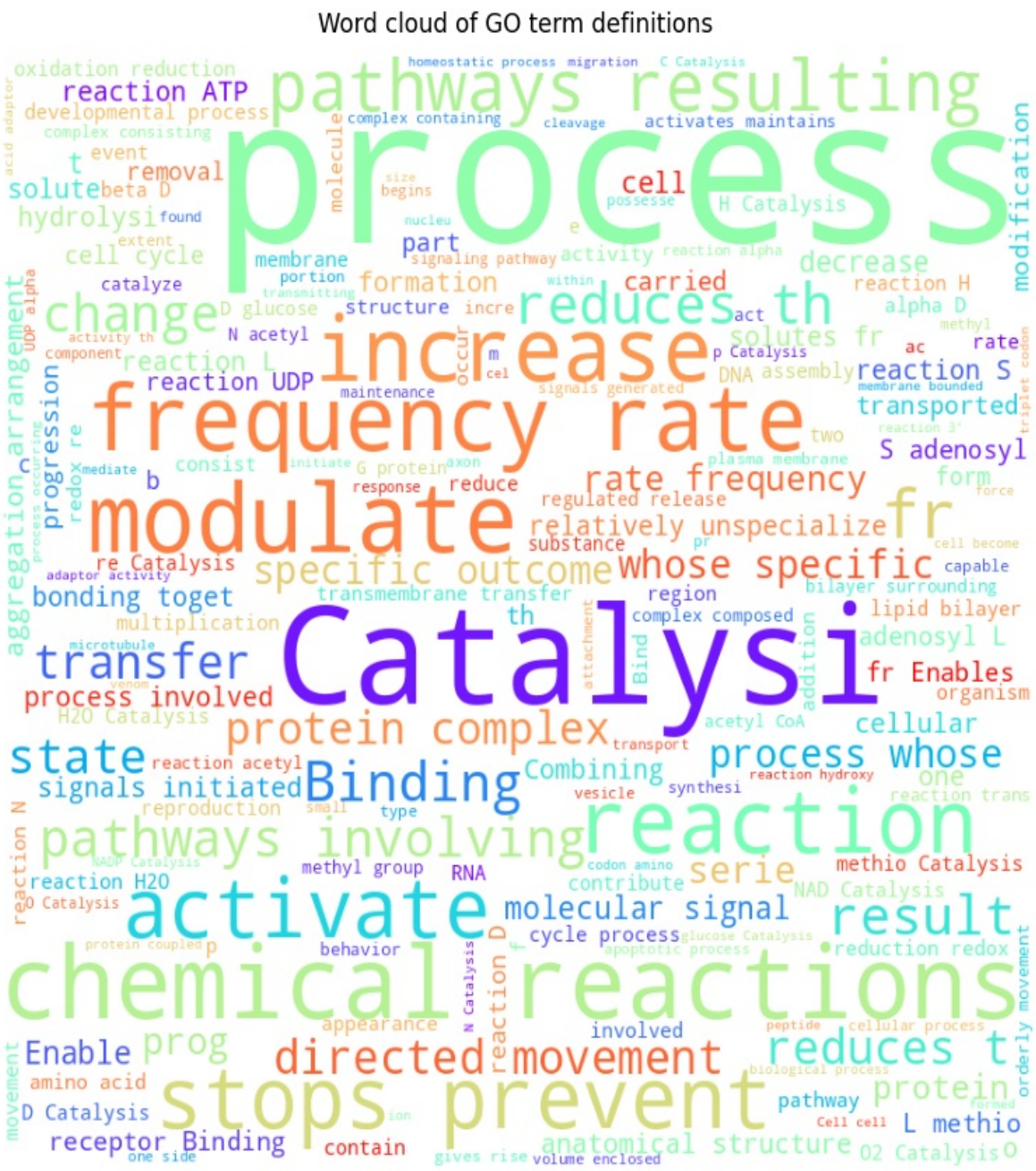
```
# print label mapping for reference
print("\nCode: label")
for code, aspect in enumerate(go_df.aspect.cat.categories):
    print(f"{code}: {aspect}")
```

```
Code: label
0: biological_process
1: cellular_component
2: molecular_function
```

We can visualize the most common terms in the dataset by creating a word cloud. This visual representation is quite helpful in identifying the most frequently occurring words in a given text.

```
from wordcloud import WordCloud

go_wordcloud = WordCloud(
    width=800,
    height=800,
    background_color="white",
    min_font_size=10,
    colormap="rainbow",
    random_state=13,
).generate(data.text.to_string())

plt.figure(figsize=(8, 8), facecolor=None)
plt.suptitle("Word cloud of GO term definitions")
plt.imshow(go_wordcloud)
plt.axis("off")
plt.tight_layout(pad=1)
plt.show()
```

Below, we can clearly see that life is all about a system of **controled catalysis**, and the information encoded in the genomes allows such organized system to be passed forward:



Dataset Creation and Splitting

To construct the dataset containing our text sequences and corresponding labels for fine-tuning the model, we will leverage the Dataset class provided by Hugging Face. Using its built-in dataset split method, we will create distinct training and testing sets. We will ensure that the sampling is **stratified by label**. This stratification is crucial as it **preserves the distribution of the GO ontologies**, ensuring a representative balance in our datasets.

```
from datasets import Dataset

# create dataset
dataset = Dataset.from_pandas(data)

# change label column to ClassLabe to allow stratification
dataset = dataset.class_encode_column("label")

# split the dataset into train (80%) / test (20%)
dataset = dataset.train_test_split(test_size=0.2, stratify_by_column="label")
```

The resulting dataset is nested dictionary containing both sets and the GO term definitions with the respective labels:
`DatasetDict({ train: Dataset({ features: ['text', 'label'], num_rows: 34598 }) test: Dataset({ features: ['text', 'label'], num_rows: 8650 }) })`

Text Tokenization

The **tokenization strategy must align with the chosen model**. This is crucial to ensure that tokens are mapped to the **same indices** presented to the model during training and that the **same special tokens** are used to denote the beginning of a text sequence and the separation between sentences. We also need to handle inputs that are longer or shorter than the input accepted by the model, or defined by us when we initialize the tokenizer.

> **Handling text input length with truncation and padding strategy**: The model accepts fixed-sized tensors for training and evaluation. To ensure that we feed the model with fixed-sized tensors, the tokenizer handle sequences of varying lengths by truncating and padding longer and shorter text sequences, respectively. If the *truncation* parameter is set to 'True,' input sequences longer than a defined length will be truncated. Conversely, if the sequences are shorter, the special padding token [PAD] will be added until the number of tokens in the input matches our requirements for the model of choice. Truncation and padding are pivotal in guaranteeing consistent input sizes, as the inputs for the model we chose are predefined fixed-size tensors (e.g., 510 tokens + 2 special tokens [CLS] and [SEP] = 512). For our **fine-tunning** task, setting the input maximum length to 150 tokens encompass the majority of the samples and expedite the tokenization process.

Classification Token [CLS] and **Separator Token [SEP]** are special tokens that provide information about the input provided to the model we choose. [CLS] marks the initiation of the sequence for the BERT model. [SEP] separates sentences in sentence-pair tasks. It aids the model in capturing the relationships between two sentences that are concatenated using [SEP]. Note the use of padding [PAD] to fill-up the input sequence to a fixed length.

In this context, we are employing the BertTokenizerFast with the tokenization strategy of the pre-trained model bert-base-multilingual-uncased to convert text to lowercase and eliminate capitalization. Lowercase inputs typically result in a smaller number of tokens and tend to generalize better for unseen text sequences during production phase. However, we can later explore the possibility of using cased inputs to evaluate model performance.

As indicated by the **boxplot** above, all GO term definitions conform to the input size requirements for fine-tuning (510 tokens). Nonetheless, we might consider limiting the input length to expedite the fine-tuning process. To proceed, we will create the dataset with training and test sets using Hugging Face's Dataset library. We will employ stratification to maintain the distribution of labels between datasets.

```
from transformers import BertTokenizerFast

# initiate tokenizer with parameters from the pre-trained model
tokenizer = BertTokenizerFast.from_pretrained("bert-base-multilingual-uncased")


# define tokenize function to be applied to the input (lowercase is applied by default)
def tokenize_function(examples):
    return tokenizer(examples["text"], padding=True, max_length=100, truncation=True)


# tokenize dataset
tokenized_dataset = dataset.map(tokenize_function, batched=True)
```

Now, our dataset contains additional stuff that will be fed to the model during training:

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 34598
    })
    test: Dataset({
        features: ['text', 'label', 'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 8650
    })
})
```

Let's see what are these new features created by the tokenizer by cheking the first sample from the train dataset:

```
print("text:", tokenized_dataset["train"]["text"][0])
print("label:", tokenized_dataset["train"]["label"][0])
print("input_ids:", tokenized_dataset["train"]["input_ids"][0])
```

```
print("token_type_ids:", tokenized_dataset["train"]["token_type_ids"][0])
print("attention_mask:", tokenized_dataset["train"]["attention_mask"][0])
```

```
text: The aggregation, arrangement and bonding together of a set of components to form a virus tail fibe

label: 0

input_ids: [101, 10103, 13353, 37910, 69649, 117, 34429, 10110, 19600, 10285, 13627, 10108, 143, 10486,

token_type_ids: [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,

attention_mask: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
```

Each feature holds specific information relevant to the tokenized dataset:

- **text**: the input text sequence.
- **label**: numerical values corresponding to the classes (i.e. ontologies) to predict:
  - 0: biological_process
  - 1: cellular_component
  - 2: molecular_function
- **input_ids**: These are the indices of each token within the sequence. These indices were generated during the model's pre-training phase and are now applied to our input.
- **token_type_ids**: This feature provides references to the sentence to which each token belongs.
- **attention_mask**: It indicates whether the token should be attended to during processing.

Fine-Tuning the Model

Continuing our progress, we will initialize the BertForSequenceClassification model and load the **weights** from the pre-trained model. Additionally, we will specify the number of target labels we aim to predict. In our case, we intend to classify whether the text corresponds to Biological Process (label = 0), Cellular Component (label = 1), or Molecular Function (label = 2). Consequently, the number of labels is set to **3**.

It's crucial to recognize that training Large Language Models can be computationally intensive, often demanding substantial hardware resources. The model we are utilizing consists of **168 million trainable parameters**, pre-trained in a semi-supervised manner. While we won't be training all of these parameters, but only a subset for our specific task, it remains a significant computational task.

> The **carbon footprint** of such large language models can be quite taxing. An additional important benefit of fine-tunning is reducing the carbon footprint when deploying custom models. This interesting seminar from The Royal Institution YouTube channel talks about it.

Depending solely on a CPU for training and fine-tuning may lead to lengthy and impractical processing times.

> Hugging Face's Trainer seamlessly handles model and data allocation between devices. If a GPU is available and correctly configured in the system, the Trainer class will utilize it.

```
from transformers import BertForSequenceClassification

model = BertForSequenceClassification.from_pretrained(
"bert-base-multilingual-uncased", num_labels=3
)
```

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(105879, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
```

```
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
            ...
            (output): BertOutput(
              (dense): Linear(in_features=3072, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
      (pooler): BertPooler(
        (dense): Linear(in_features=768, out_features=768, bias=True)
        (activation): Tanh()
      )
    )
    (dropout): Dropout(p=0.1, inplace=False)
    (classifier): Linear(in_features=768, out_features=3, bias=True)
)
```

Above we can see the BERT model architecture. Note that the **output** layer is a fully-connected neural network that take the embeddings of the previous layer and yields **logits** of size 3. Logits typically refers to the raw scores generated by the classifier before applying a **softmax** function to return the probabilities of each label (or *class*). These logits are often used in multi-class classification problems to represent the unnormalized prediction scores for each class.

Fine-Tuning with Trainer

Our next step involves fine-tuning the model using the Trainer class, a specialized tool designed to optimize the training of Hugging Face's models. Subsequently, we will explore fine-tuning the model using native PyTorch methods.

To configure the Trainer effectively, it's essential to set the **hyperparameters** utilizing the TrainingArguments and specify the evaluation metrics through the evaluate module. This strategic setup allows us to closely monitor the training progress and evaluate the model's performance after the fine-tuning process.

Metrics

Given that the Trainer lacks an automatic performance evaluator during training, it's crucial to integrate metrics and pass them to the Trainer object through TrainingArguments.

For our performance assessment, we will employ the ROC/AUC score. To facilitate this, we will create a function to convert **predictions** (probabilities ranging from 0 to 1, inclusive, obtained from the softmax function) into **logits** (the raw output from the model with unnormalized scores because all transformers models return logits. By incorporating this custom metric, we can effectively evaluate the model's performance during the fine-tuning process.

```
from transformers import TrainingArguments
import evaluate

# load roc_auc metric
metric = evaluate.load("accuracy")

# convert preds --> logits
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)

# set evaluation for training
train_args = TrainingArguments(
    output_dir="test_trainer",
    evaluation_strategy="epoch",
)
```

```
train_args = TrainingArguments(output_dir="test_trainer")
```

To reduce the training duration and make it more suitable for a tutorial, we will fine-tune the model using a smaller subset of the training data, reserving 20% of it for validation purposes.

```
small_train_dataset = tokenized_dataset["train"].shuffle(
    seed=42).select(range(2000))
small_eval_dataset = tokenized_dataset["test"].shuffle(
    seed=42).select(range(1000))
```

Finally, we can initiate the model training process. The Trainer class, provided by Hugging Face, simplifies the training procedure to just a few lines of code. Given all the information we've covered so far, the parameters to be passed to the Trainer become quite self-explanatory.

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=train_args,
    train_dataset=small_train_dataset,
    eval_dataset=small_eval_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)

trainer.train()
```

Here is the output stats of the epoch=3. Fine-tuning a pre-trained model usually requires just a few epochs to achieve high accuracy (ROC/AUC score = 0.986). This is because the deep features of language semantics were already learned during **pre-training**:

```
{'eval_loss': 0.08846566081047058, 'eval_accuracy': 0.986, 'eval_runtime': 6.3565, 'eval_samples_per_sec
{'train_runtime': 162.5061, 'train_samples_per_second': 36.922, 'train_steps_per_second': 4.615, 'train_
```

Fine-Tuning with Native PyTorch

Having explored the conveniences of Hugging Face's classes and models, it's valuable to delve into the internal workings of the training loop. Understanding these mechanisms equips you to make refinements that could enhance the model's performance or render it more lightweight for deployment.

In this section, we will fine-tune our BERT model using native PyTorch. This approach entails implementing the training loop and creating the dataloader responsible for supplying batches of examples during training. By proceeding with the exercise of fine-tunning the model using native PyTorch, we gain a deeper understanding of the training process, fostering the ability to make tailored adjustments to achieve the desired model's performance in our classification task.

But first we might want clean-up GPU's memory and delete the *model* and *trainer* objects to release space and set the model to the initial state. Optionally, we can restart the notebook.

```
del model
del trainer
torch.cuda.empty_cache()
```

Dataloader

To facilitate the training process, we need to configure the PyTorch DataLoader, which combines the dataset structure and a sampler to provide batched chunks of training data to the model during training. The DataLoader is optimized for memory occupancy and speed, making it ideal for our needs. However, before we proceed, we must make some modifications to the tokenized dataset to align it with the input requirements of the model:

1. Remove the *text* column from the dataset since the model does not accept text as input.
2. Rename the *label* column to *labels* as the model is preset with this specific column name.

3. Instruct the dataset to return PyTorch tensors.
4. Select only a small portion of the dataset for fine-tuning the model, similar to what we did before.
5. Instantiate the data loaders for training and test sets, shuffling the training samples but not the evaluation set. Also, batch sizes tends to be small during training and fine-tunning of such large models due to memory constraints.

```
from torch.utils.data import DataLoader

# remove the text column
tokenized_dataset = tokenized_dataset.remove_columns(["text"])

# rename the label column
tokenized_dataset = tokenized_dataset.rename_column("label", "labels")

# set torch tensors as the dataset output
tokenized_dataset.set_format("torch")

# create smaller training and test subsets
small_train_dataset = tokenized_dataset["train"].shuffle(
    seed=42).select(range(2000))
small_eval_dataset = tokenized_dataset["test"].shuffle(
    seed=42).select(range(1000))

# configure torch data loader (eval set don't should be shuffled)
train_loader = DataLoader(small_train_dataset, shuffle=True, batch_size=8)
eval_loader = DataLoader(small_eval_dataset, batch_size=8)
```

This is the expected output, without the text input. The other features remains the same:

```
Train:
 Dataset({
    features: ['labels', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 2000
})

Test:
 Dataset({
    features: ['labels', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 1000
})
```

Optimizer and Learning Rate

The optimizer adjusts the weights of the network during training to minimize the **loss** computed by a loss function, i.e., minimize the discrepancy between the input and output of the model. Some optimization algorithms, such as RMSProp, Adam and AdamW, dynamically adapts the learning rate of each network parameter based on the gradient descent previous magnitudes.

The recommended optimizer to fine-tune BERT models is AdamW, a variant of Adam that implements weight decay to improve model's convergence and generalization power by preventing overfitting. We will implement an scheduler to adapt the learning rate along the training process.

> Remember that **overfitting** happens when the model achieves high scores in predicting the training data output but has **little generalization power for unseen data**.

First things first, let's instantiate the model with pre-trained weights and move it to the GPU memory:

```
import torch
from transformers import BertForSequenceClassification

model = BertForSequenceClassification.from_pretrained(
    "bert-base-multilingual-uncased", num_labels=3
)

# check GPU availability
device = torch.device(
```

```
    "cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)
```

And now instantiate the optimizer and the learning rate scheduler:

```
from torch.optim import AdamW
from transformers import get_scheduler

# implement optimizer with small learning rate (recommended for BERT)
optimizer = AdamW(model.parameters(), lr=5e-5)

# implement scheduler
num_epochs = 3
num_training_steps = num_epochs * len(train_loader)  # total number of batches

lr_scheduler = get_scheduler(
    name="linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)
```

Training Loop

Now we can train, I mean, fine-tune the model. The training loop controls the training process by allocating the batches of data to the model's device location and trigger the computation of the loss and optimization steps. In the training loop we also specify the number of epochs that we will be training our model.

To monitor the training progress, we can set a progress bar using the handy tqdm library. We can just wrap any interable with tqdm that it will take care of the rest.

```
# the total number of batches going through the model
progress_bar = tqdm(range(num_training_steps))

# activate the training mode
model.train()

# the loop itself
for epoch in range(num_epochs):
    for batch in train_loader:
        # moves the data to the specified device (8 samples per batch)
        batch = {key: value.to(device) for key, value in batch.items()}
        # unpack batch dict and get the outputs to compute loss
        outputs = model(**batch)
        # extract the loss attribute from the output object
        loss = outputs.loss
        # compute gradients (derivatives)
        loss.backward()
        # update the weights using the computed gradients
        optimizer.step()
        # adjust the learning rate after each pass
        lr_scheduler.step()
        # reset the gradients because the weights are updated
        # in each pass using only the gradients of the batch.
        optimizer.zero_grad()
        progress_bar.update(1)
```

Evaluation

To assess the performance of the trained model, it's essential to define the metric (just as we did when fine-tuning the model using the Trainer class) and create the evaluation loop. During evaluation, we provide batches to the model, accumulate predictions in the defined metric's attributes, and calculate the final score.

Before evaluation, we switch the model to evaluation mode by invoking the model's `eval()` method. This action alters the

behavior of certain layers that behave differently between training and evaluation, such as Dropout and Batch Normalization layers. It also informs the model not to compute gradients. Here's why:

1. **Dropout**: Dropout randomly deactivates a fraction of neurons during training to prevent overfitting. In evaluation mode, this operation is unnecessary because we aim to predict all samples consistently.
2. **Batch Normalization**: During training, each batch is normalized independently. In evaluation mode, we accumulate batches and perform normalization at the end considering the population statistics (mean and variance) learned during training.
3. **Gradient Computation**: Since no weights are updated during evaluation mode, disabling gradient computation saves memory and processing time. We also use the PyTorch context manager `torch.no_grad()` to disable gradient computation in the evaluation loop.

Setting the model to evaluation mode ensures that it behaves consistently and produces reliable predictions during the evaluation process.

```
import evaluate

# define metric
metric = evaluate.load("accuracy")

# set the model to evaluation mode
model.eval()

# evaluation loop
for batch in eval_loader:
    batch = {key: value.to(device) for key, value in batch.items()}

    with torch.no_grad():
        outputs = model(**batch)

    logits = outputs.logits
    predictions = torch.argmax(logits, dim=-1)
    metric.add_batch(predictions=predictions, references=batch["labels"])

print(f"Accuracy: {metric.compute()['accuracy']}")


Accuracy: 0.989
```

Using the Fine-Tunned Model for Predictions

We've fine-tunned our model to understand the specific semantics of GO terms definitions regarding to which aspect the sentence is more likely to belong: Biological Process (BP), Cellular Component (CC) and Molecular Function (MF). An interesting possible application is using the model to classify any sentence regarding these aspects. One could ask: What is the predominantly GO aspect in a scientific text or sentence?

We could determine the predominant GO aspect (biological process, celullar localization or molecular function) of single sentence and a paper abstract on molecular biology.

Let's start by examining a single sentence sample. Remember that this is just a sample. Predictions make sense within the context they were fine-tuned for.

> Just for the sake of curiosity, this is the **NASA's definition of life** when searching for life elsewhere in the Universe. It is the more concise we can think of.

```
sample = "Life is a self-sustaining chemical system capable of Darwinian evolution."
```

As we did before, we tokenize the text sequence using the same parameters that we've used to configure the tokenizer for fine-tunning the model:

```
# tokenize input
inputs = tokenizer(
```

```
    sample, max_length=100, truncation=True, padding=True, return_tensors="pt"
).to(device)

# switch model to evaluation mode
model.eval()

# get logits
with torch.no_grad():
    logits = model(**inputs).logits

# calculate probabilities from logits
probs = torch.nn.functional.softmax(logits, dim=1)
prediction = torch.argmax(probs, dim=1).item()

# print label probabiliteis
print("Probabilities:")
for code, aspect in enumerate(go_df.aspect.cat.categories):
    print(f"{aspect}: {probs[0][code]:.3f}")

print(f"\nPredicted Ontology: {go_df.aspect.cat.categories[prediction]}")
```

```
Probabilities:
biological_process: 0.868
cellular_component: 0.107
molecular_function: 0.025

Predicted Ontology: biological_process
```

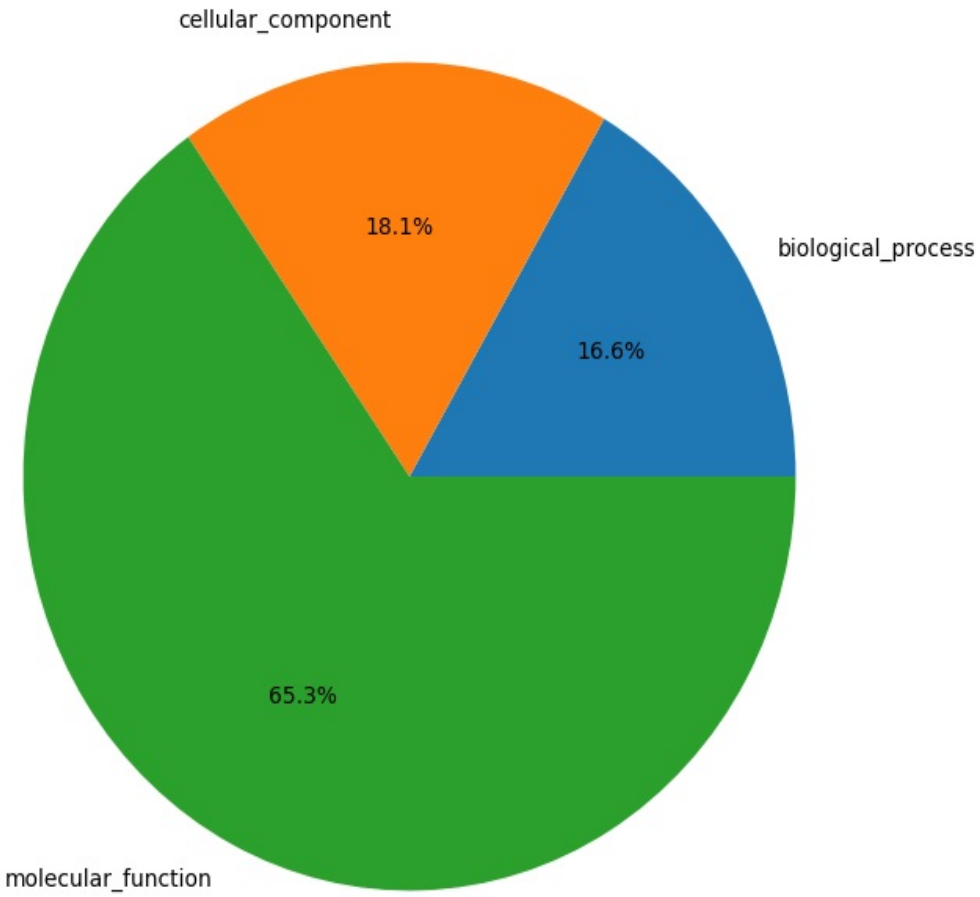
We can also make predictions for a longer input such as an article abstract, or even the whole article, but first we need to split the text into sequences. For this, we can use the punkt module from Natural Language Toolkit (NLTK). The code below download the required punkt tokenizer data (only needed once):

```
import nltk

nltk.download("punkt")
```

The sample text is included in the repository, but you can use any other you like:

```
# load sample text
file = open(home_dir.joinpath("data/sample_text.txt"), "r").read()

# split sample text into sentences and put into a dataframe
sentences_list = nltk.tokenize.sent_tokenize(file)
sentences_df = pd.DataFrame(columns=["sentence"], data=sentences_list)
print(f"Number of sentences: {len(sentences_list)}")

# switch to evaluation mode
model.eval()

# empty list to store temporary dictionaries with samples' predictions
data_list = []

# iterates over the sentences' list, tokenize, predict and append results
for sample in sentences_list:
    inputs = tokenizer(
        sample,
        max_length=100,
        truncation=True,
        padding=True,
        return_tensors="pt",
    ).to(device)

    with torch.no_grad():
        logits = model(**inputs)
```

```
        prediction = torch.nn.functional.softmax(logits.logits, dim=1)

    # get predictions out of GPU's memory, convert to list for appending
    prediction_list = prediction.cpu().numpy().tolist()[0]

    data_dict = {"sentence": sample}

    # update results dictionary with predictions for every sentence
    # zip() yields tuples containing the elements from the same indices
    # in the iterables passed as parameters until the shortest iterable
    # is exhausted.
    data_dict.update(
        {
            category: prob
            for category, prob in zip(go_df.aspect.cat.categories, prediction_list)
        }
    )

    data_list.append(data_dict)

# create dataframe with predictions
probs_df = pd.DataFrame(data_list)

# calculate mean for each aspect in the text
probs_sample = probs_df.mean(numeric_only=True)


probs_df = probs_df.style.format(
    {
        "biological_process": "{:.2%}",
        "cellular_component": "{:.2%}",
        "molecular_function": "{:.2%}",
    }
)

# print / plot results
probs_sample_results = probs_sample.map(lambda x: "{:.2%}".format(x)).sort_values(
    ascending=False
)

# save results
home_dir.joinpath("output/").mkdir(parents=True, exist_ok=True)
probs_df.to_excel(home_dir.joinpath("output/results.xls"), index=False)

# visualize results
print(probs_sample_results)

plt.figure(figsize=(8, 8))
plt.pie(probs_sample, labels=go_df.aspect.cat.categories.to_list(), autopct="%1.1f%%")
plt.show()
```

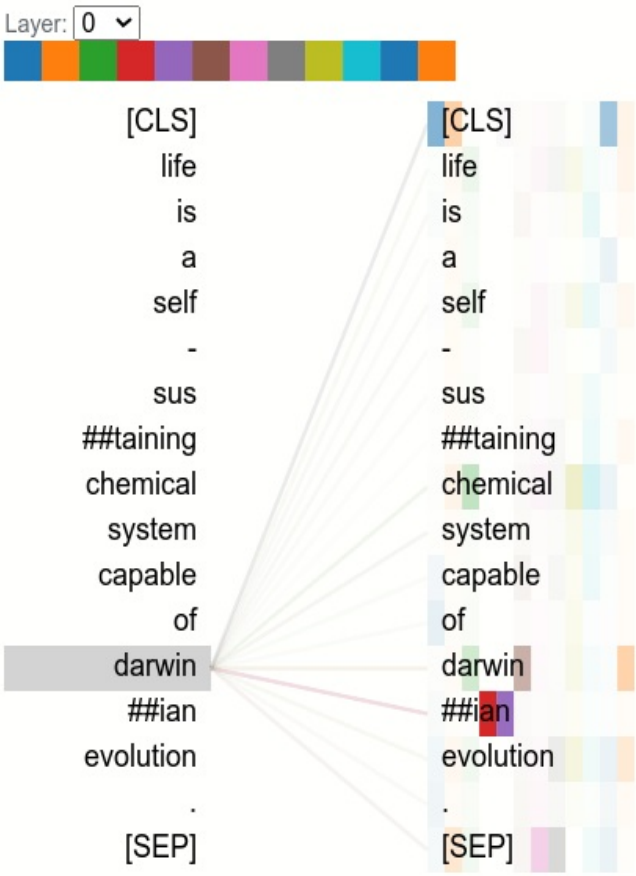


	sentence	biological_process	cellular_component	molecular_function
0	Acts as a tumor suppressor in many tumor types; induces growth arrest or apoptosis depending on the physiological circumstances and cell type.	0.33%	0.08%	99.58%
1	Involved in cell cycle regulation as a trans-activator that acts to negatively regulate cell division by controlling a set of genes required for this process.	64.13%	12.42%	23.45%
2	One of the activated genes is an inhibitor of cyclin-dependent kinases.	1.94%	96.16%	1.90%
3	Apoptosis induction seems to be mediated either by stimulation of BAX and FAS antigen expression, or by repression of Bcl-2 expression.	99.73%	0.21%	0.05%
4	Its pro-apoptotic activity is activated via its interaction with PPP1R13B/ASPP1 or TP53BP2/ASPP2 (By similarity).	1.04%	0.47%	98.49%
5	However, this activity is inhibited when the interaction with PPP1R13B/ASPP1 or TP53BP2/ASPP2 is displaced by PPP1R13L/iASPP (By similarity).	0.42%	0.09%	99.49%
6	In cooperation with mitochondrial PPIF is involved in activating oxidative stress-induced necrosis; the function is largely independent of transcription.	8.17%	2.22%	89.60%
7	Prevents CDK7 kinase activity when associated to CAK complex in response to DNA damage, thus stopping cell cycle progression.	0.47%	0.08%	99.45%
8	Induces the transcription of long intergenic non-coding RNA p21 (lincRNA-p21) and lincRNA-Mkln1.	0.46%	0.12%	99.42%
9	LincRNA-p21 participates in TP53-dependent transcriptional repression leading to apoptosis and seems to have an effect on cell-cycle regulation.	5.37%	87.35%	7.28%
10	Regulates the circadian clock by repressing CLOCK-BMAL1-mediated transcriptional activation of PER2	0.46%	0.07%	99.47%

What Are You Paying Attention To? Find Out with BERT Visualizer

Some days, you stumble upon beautiful and useful tools. Today, that gem is BertViz, an interactive tool for visualizing attention in transformer models. In the code below, we visualize the attention heads in a single layer:

```
from bertviz import model_view, head_view

# sample text
sample = "Life is a self-sustaining chemical system capable of Darwinian evolution."

# tokenize inputs
inputs = tokenizer.encode(sample, padding=True,
                          truncation=True, return_tensors="pt")

# evaluate and return attentions and corresponding tokens
outputs = model.cpu()(inputs, output_attentions=True)
attention = outputs[-1]
tokens = tokenizer.convert_ids_to_tokens(inputs[0])

head_view(attention, tokens)
```



This is just a taste of **model interpretability**, a crucial topic in machine learning that aims to facilitate the human interpretation of results from machine learning models. This concept is key to obtain insights on the inner mechanisms of the studied phenomena, allowing us to understand model's output. In future posts, we will delve a bit more in this hot topic.

In the **next post**, we'll dive into **text embeddings** and how to use what we've just learned to **automatically annotate newly discovered genes/proteins** according to Gene Ontology.