Fine-tuning Pretrained Language Models for Biomedical Tasks

Student ID : 31334679

Name: Jirarote Jirasirikul

Masters of Data Science

FIT5128 - Minor Thesis Final

Supervisors:

Gholamreza (Reza) Haffari,

Ehsan Shareghi

A thesis submitted for the degree of Master of Data Science at Monash University in 2021

Word count for Part 2 (excluding references): 6090

Acknowledgements

I would like to express my deepest gratitude to my thesis supervisors Dr. Gholamreza Haffari and Dr. Ehsan Shareghi for their unwavering support and guidance throughout the entire project, which I came to learn about so many new things in Natural Language Processing (NLP).

I am profoundly grateful to have conducted my research under their supervision from start to end. Without their dedication, this would not have been possible.

Thank you,
Jirarote Jirasirikul (JJ)

Table of Contents

Part 1: General Literature Review

1. Introduction

With the rapidly growing amount of data, digitalization of documents is no longer a barrier for Data Scientists. Every report is now provided in an electronic form or physical paper with electronic copy, this includes our personal medical document and treatment history. These information are stored somewhere within databases and we could retrieve them by queries when necessary. Our problem became a paradox as we are now overflowing with these data. Humans have limited capacity for concentration, which they commonly use for information that they are interested in. Regrettably, lots of information that is potentially valuable is often overlooked. Natural Language Processing has become important because we could use computation to help manage these data and assist in detecting interest and unusual patterns. These bring missout beneficial data into our attention and lessen precious information unattended.

In health services, these problems are no less vulnerable. As a matter of fact, we could say they are more vital to not leave any data unnoticed because this could lead to a life-death situation. We see these as a gap and opportunity to assist these devoted medical staff in identifying potential life-threatening diseases in patients and bring them into attention to get earlier treatment. One of the reports that medical staff commonly use is a radiology report. This report is a summarization of the assessment performed by a radiologist in order to address general practitioner concern on the patient. In order to conclude evaluation, multiple medical procedures were performed to detect and diagnose, some relevant and some not. However, radiologist tasks are only to address general practitioner focus. Out of those assessments some would be picked and pointed to diagnose immediate patient illness. The rest remain in the report as supporting documents. However as mentioned earlier, general practitioners have their hands full on concentrating treatment for immediate illness. Unfortunately, some information might have been overlooked.

Our goal is to use Machine Learning to figure out some common patterns using supervised and unsupervised learning methods on these radiology reports. There are two main tasks that would be required in order to address this problem: (1) Understanding medical reports which are embedded with specialized domain knowledge. (2) Build prediction models to calculate probability of patients infected with fungal disease. These tasks are connected and contribute to quality prediction performance. We will face some foresee challenges in this thesis as we are processing medical information: (1) data privacy of medical information (2) medical domain specialize corpora (3) precision and accuracy of the prediction on disease.

This thesis focuses on Text classification using the state-of-the-art of Natural Language Processing to handle radiology reports and detect unique patterns that could show signs of infection. The prediction results will assist General Practitioners to be aware of additional complications that may occur in their patients. Conceivably, this will reduce beneficial data misout to the bare minimum.

2. Substantive Literature Review

Natural language processing (NLP) is a branch of study in Machine Learning that helps computers understand, interpret and process human language. This involves a range of computational techniques for analyzing and representing naturally occurring texts (Liddy, n.d.; Louis, 2020). Conventional natural languages are typical unstructured information and NLP aims to extract useful information which represents those raw data by feature engineering (Z. Liu et al., 2020).

In this literature review, our goal is to look back into the history of text representation methods in natural language processing, understand the benefits and limitations of each improved method, in order to see the possibility to adapt these techniques into medical use cases. By doing so, we hope to find a better way to extract valuable information from medical reports that may be left unnoticed and could be extremely beneficial to people's life.

A typical machine learning system, that builds to assist in given tasks such as classification or forecasting, are usually follow consists of three components:

Machine Learning = Data representation + Objective + Optimization

Data representation is one of the core components to build a good model. In NLP, our data is usually in a form of Text, Word or Sentence representation. The Figure 1 diagram below highlights some of the significant algorithms created along the NLP journey (*Evolution of Natural Language Processing*, 2020). We will explore, summarize and argue throughout the following subsection.

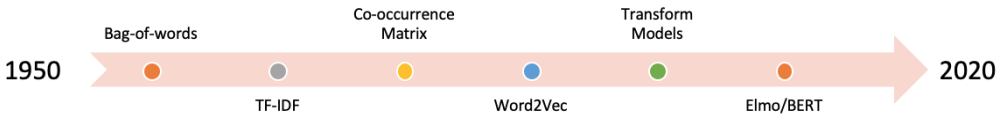


Figure 1: Evolution of NLP's Text representation

2.1 General Text Representation

Computer ("Computer," 2021) is an electronic device for storing and processing data in binary form. This means that in order to make computers understand the meaning of human language. They need to transform language text 'words' into a form of 'numbers'. Generally, computers automatically convert each character of the alphabet symbol into numbers called ASCII numbers (Hieronymus & Laboratories, n.d.). Although, this does not give a representation of the words rather than an alphabet, using the same methodology could still be applied. By representing words as numbers, we could count occurrences of each word in passage or documents (*Evolution of Natural Language Processing*, 2020; Zhang et al., 2010) and perform statistical inference. This method is called the "**Bag-of-Words**" (BoW) representation which is a successful and popular approach for document categorization where word distribution could determine a document's topic. However, BoW representation

has disadvantages as it does not consider type words such as common words or stopwords that appear frequently but has less valuable meaning. Term frequency-Inverse document frequency (TF-IDF) has been introduced to address this problem by putting weight to important words that might occur less frequently shown in Figure 2. Nevertheless, these approaches disregard valuable information on the position of words in a context.

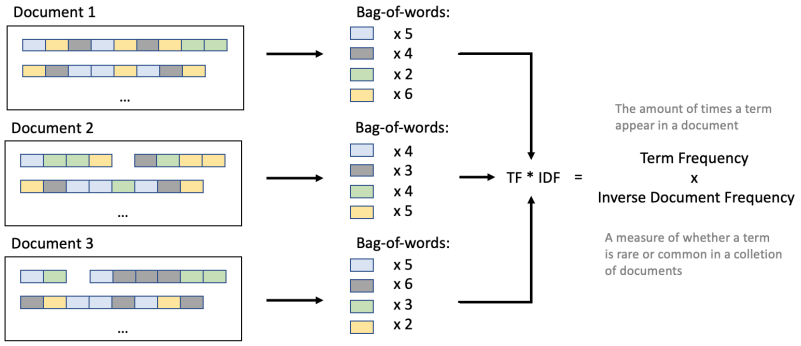


Figure 2: Illustrate a construction of Bag-of-words and TF-IDF from documents

Alternatively, words can also be represented in a form of vector instead of number. This text representation is called **one-hot-encoding** which will give an index of words in the dictionary (*Andre Ye*, n.d.)

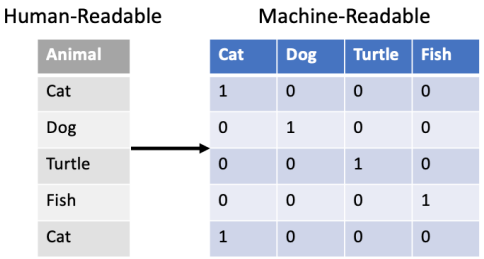


Figure 3: One-hot-encoding from word to vector of index in dictionary

A **Co-occurrence Matrix** (Manning et al., 2008; Vargas, 2017) was created by analyzing the context in which a word is used and taking the neighboring words of each word into account. This technique generated word embeddings that track context of the word by using large matrices, Figure 4. As a consequence, this method requires a large amount of memories which are not preferable.

Sentence : I love Programming. I love Math. I tolerate Biology.

	I	love	Programming	Math	Tolerate	Biology	.
I	0	2	0	0	1	0	2
love	2	0	1	1	0	0	0
Programming	0	1	0	0	0	0	1
Math	0	1	0	0	0	0	1
Tolerate	1	0	0	0	0	1	0
Biology	0	0	0	0	1	0	1
.	1	0	1	1	0	1	0

Figure 4: Co-occurrence matrix build from sentence by counting neighbor of each word

As NLP progressed, word embedding became a frequent norm to achieve state-of-the-art results in machine learning. Researchers seek to find an improvement of embedding context that is relevant into the word so that semantic meaning of the word could be extracted. **Word2Vec** was introduced in late 2013 (Mikolov et al., 2013) by generating vectors that could represent the meaning of words after going through a large scale corpus. The simplest part of this idea is by looking for previous n-1 words from the focus word and calculating the probability of the possible word that fit the focus word, Figure 5. There are 2 algorithms that infer from this idea which is CBoW and skip-gram. While CBoW is looking at n-1 words to try identify the focus words. Skip-gram, on the other hand, is trying to find the highest possibility of words that would come before or follow the focus word. Hence after massive learning of corpus, it became possible to form a vector that represents each word.

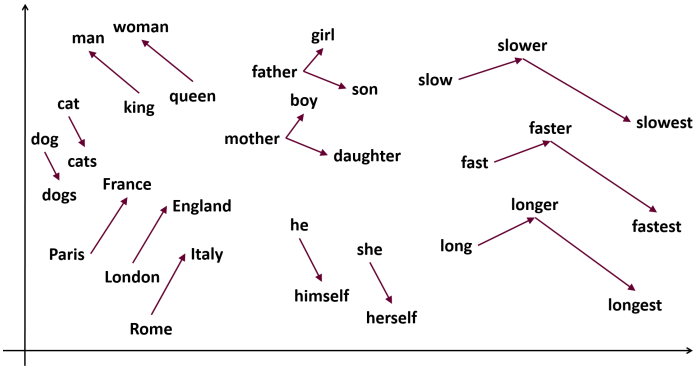


Figure 5: Example of words in 2D dimensions of Word2Vec's vectors (Pal, 2019)

Classic example that shows how Word2Vec works and how it is embedded with semantic meaning are:

King - Man + Woman = Queen

However, Word2Vec accuracy is not as high as it claims (Church, 2017, p. 2). There are words that have similar vector with comparable probability but represent totally different meanings for instance in sample context example, i.e. Monarch, Princess, Prince, could lead to different meanings of the sentence. Nevertheless, Word2Vec is the most basic and standard text representation that is the most well known up-to-date. This is because the Word2Vec approach is simple and easily accessible which overrules its inaccuracy and incorrectness. The Word2Vec contributes a large impact to the NLP fields and fundamentals to text representation.

Meanwhile, the Neural network based model has started to become an increasingly popular and dominant approach to tackle NLP tasks, as they produce state-of-art results (Conneau et al., 2017; Joulin et al., 2016). They tend to be relatively slow when performing in both train and test datasets. Facebook AI researcher has developed the **FASTTEXT** approach (Mikolov et al., 2013), which was inspired by enhanced efficiency in word representation learning, using a linear model with rank constraint and loss approximation on word representation. FASTTEXT allows models to train billions of words within ten minutes on word representation, yet, achieved performance on par with state-of-the-art evaluated in 2 NLP tasks.

The concept of pre-train word representations became key components in natural language processing nowadays modelling (M. E. Peters et al., 2018; Vaswani et al., 2017). But because of ambiguity, word tokens could have various meanings. Pre-train word representation should not only learn the meaning of the word itself but also need to learn its context, which drives pre-train contextualized word representation. Ideally, this kind of word representations should be able to contain (1) characteristic of words (e.g. syntax and semantics) (2) application in linguistic context. Therefore, embedding these attributes into word representation is challenging. Embeddings from Language Model or **ELMo** (M. Peters et al., 2018) are presented as methods to derive word representation vectors from the context using Long-short term memory (LSTM) recurrent neural networks. Moreover, ELMo is trained bi-directional to be able to get word representation that covers both prior and after meaning of the word on a large text corpus. ELMo representations are deep because their outputs are vectors stacked that come from all internal layers of the Bi-LSTM. ELMo have outperformed previous representations and work extremely well in practice.

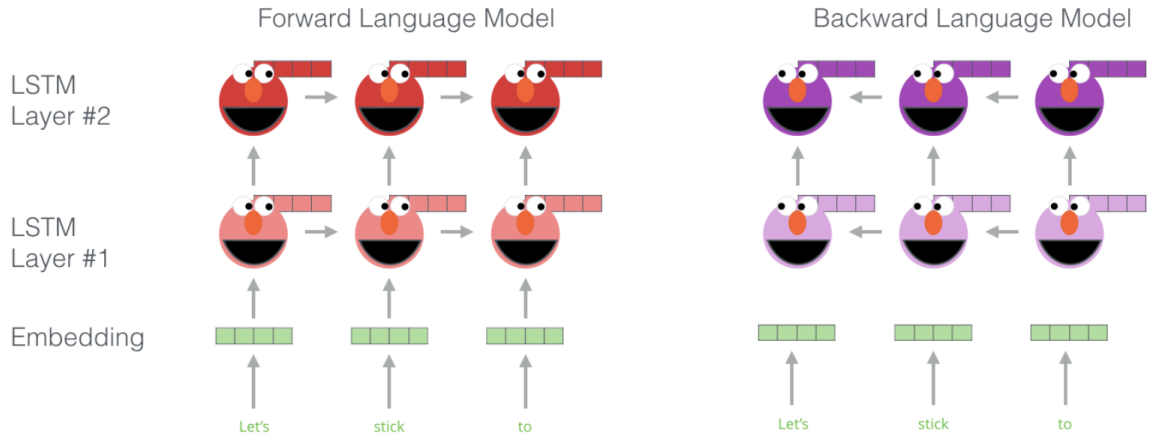


Figure 6: Illustrate on ELMo's LSTM hidden layers within Forward and Backward Language Model (Alammar, n.d.)

$$\sum_{k=1}^{N} \Big(\log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \Big)$$

Figure 7: Jointly maximize likelihood equations of ELMo's bi-LSTM forward and backward combined (M. Peters et al., 2018)

In 2017, **The transformer** paper about "attention is all you need" was published (Vaswani et al., 2017; Radford et al., n.d.). This comes with the fact that the attention mechanism deals with long-term dependencies better than LSTM, which could only recurrent back with limited window-size. Additionally, the transformer is using an encoding-decoding technique that is better with handling machine translation, Figure 8. By assigning weight for all its 'relevance' or 'attention' to each token and having each 'attention head' focus on different things, make transformers be able to outperform previous word representation. Moreover, the transformer could be trained significantly faster than LSTM architectures.

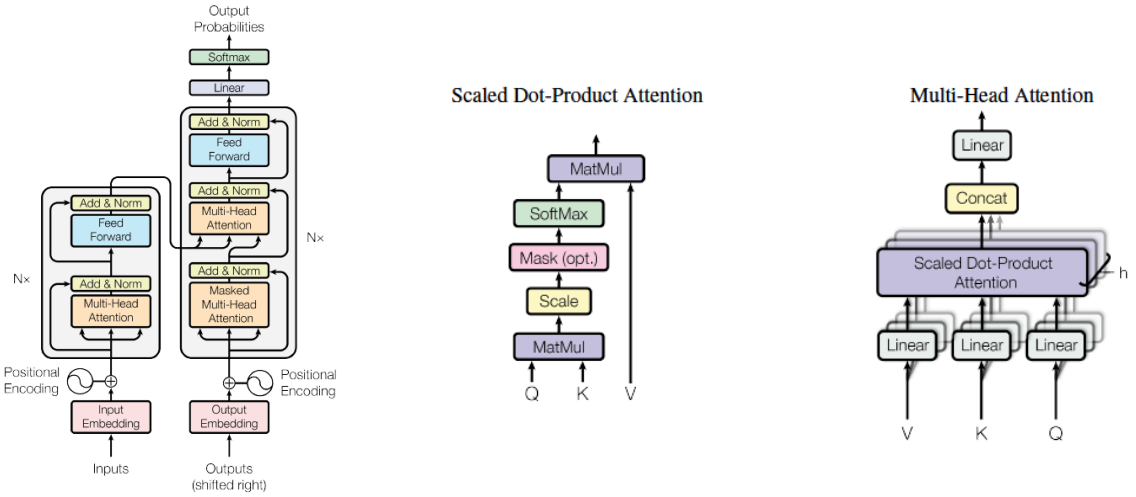


Figure 8: (left) High-level Transformer model architecture
(middle) Scaled Dot-Product Attention module
(right) Multi-Head Attention consists of several attention layers running in parallel.
(Vaswani et al., 2017)

Recently in 2019, based on the work of ELMo and The transformers, researchers have taken these two method advantages, combined them together and created **BERT** (Bidirectional Encoder Representations from Transformers), which is our current state-of-the-art (Devlin et al., 2019). Given that ELMo has the advantage of being able to capture bidirectional meaning while the transformer performs better on capturing dependencies of each token using attention mechanism. Similarly to all other pre-train models, BERT is trained on a large corpus with provided 2 pre-train models with different size $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$ for easy appliance.

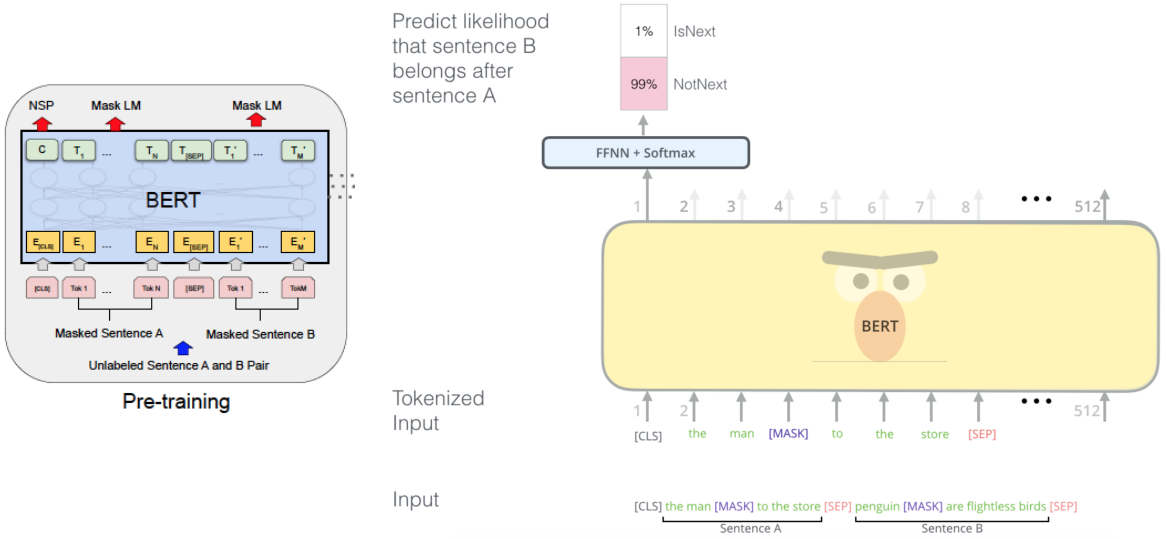


Figure 9: (left) BERT unsupervised Mask LM pretraining, (right) Illustrate BERT model in classification.
(Alammar, n.d.; Devlin et al., 2019)

BERT is the most recent state-of-the-art model. We will look into more detail of its architecture. Original BERT pre-trained using 2 unsupervised tasks called Masked Language Model (MLM) and Next sentence prediction (NSP). However, later research proves that NSP actually hurts its performance because the model is not able to learn long-range dependencies (Y. Liu et al., 2019). MLM was done by masking the token with [MASK] token in each sequence at random and making the model try to predict the correct word. This approach allows machines to capture important meaning using attention mechanisms. NSP, on the other hand, was simply adding a [CLS] token at the start of the paragraph and using [SEP] to separate between different sentences. This allows BERT to understand the relationship between 2 sentences. RoBERTa (Y. Liu et al., 2019), which carefully studies the effects of BERT hyper parameters, shows that removing NSP during pre-train matches or slightly improves downstream task performance. Nevertheless, BERT is a Language Model that encodes sentences and words into its representation which later on can be used in other models. For example: Text classification using Feed Forward Neural Network (Figure 9 - right).

Compared to the start of NLP journey, we have come so far on text representation that we have proven that contextualized word representation is better than non-contextualize word representation. Nowadays, we most likely use Language Model to represent our information before performing any modeling on-top, even though it is hardly recognized in its original form after transformation. All previous research has proven that somehow these word representations preserve characteristics of word's semantics and context because the result enhances prediction accuracy in various NLP tasks, i.e. Named Entity Recognition, Question answering, Sentiment analysis, and Text classification (*Tracking Progress in Natural Language Processing*, n.d.). This leads further on our topic to getting state-of-art performance in a more specifically medical domain NLP task.

2.2 Biomedical Text representation

In this section, we will look specifically toward text representation in the biomedical domain. Biomedical is a domain specific field that has its word distribution shifted from general domain corpora (Lee et al., 2019). Although NLP advancements yield a promising result in the general domain, applying to the biomedical text domain still remains as a challenge because of its own linguistic characteristics that are unique from the general text.

Now that we understand the history of NLP's text representations, we will focus on the recent state-of-the-art language model BERT. Our goal is to find the possibility of customization of BERT into a domain specific task. Regardless of our literature review that focuses on BioMedical Data, transferring knowledge of models from general corpora to a specific domain have a record of proven results. For example: SciBERT (Beltagy et al., 2019) and FinBERT (Yang et al., 2020) extend their corpora into the science and the finance domains. They share similar motivation that the general corpora lack quality to represent word distribution within each domain (Beltagy et al., 2019; Yang et al., 2020). Although using general corpora BERT could achieve decent results, compared with BERT that perform additional training on specific domains still show significant difference. Additional training on the Pre-train BERT model shows an improvement when handling domain specific tasks.

In the biomedical domain, **BioBERT** (Lee et al., 2019)**, ClinicalBERT** (Alsentzer et al., 2019) **and BlueBERT** (Peng et al., 2019) were driven by similar reasons. Their motivations are to improve BERT performance in handling BioMedical NLP tasks. They all perform additional training on pre-train BERT to learn large-scale BioMedical corpora. **BioBERT** approach initializes setting from general BERT and training additionally on medical corpora of PubMed abstracts (*PubMed*, n.d.) and PMC full-text articles (*Home - PMC - NCBI*, n.d.).

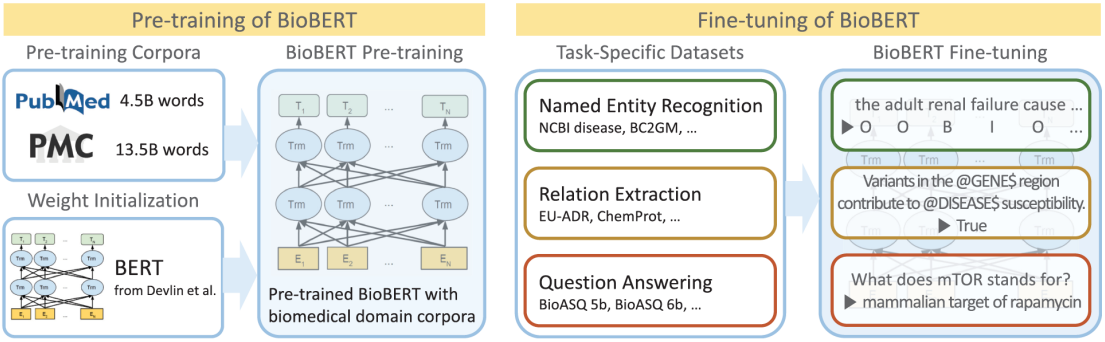


Figure 10: Overview of the pre-training and fine-tuning of BioBERT (Lee et al., 2019)

However, they were trained in 3 different ways on corpus for comparison, PubMed, PMC and PubMed+PMC, without changing the default architecture of general BERT. Moreover, BioBERT fine-tunes the model on-top to perform 3 specific NLP Tasks i.e Named Entity Recognition, Relation Extraction and Question Answering. All of these test results surpass original BERT on the same task. BioBERT claims that pre-training BERT on biomedical corpora is crucial and they provided their extension of BERT pre-trained models as open-source (https://github.com/google-research/bert). The interesting part within BioBERT paper is that training on more dataset doesn't guarantee a better performance. More specifically, training on only 'PubMed' can also perform better in some tasks, which we could see from Table 1.

Table 1: Biomedical question answering test results

			BERT	BioBERT v1.0			BioBERT v1.1
Datasets	Metrics	SOTA	(Wiki + Books)	(+ PubMed)	(+ PMC)	(+ PubMed + PMC)	(+ PubMed)
BioASQ 4b	S	20.01	27.33	25.47	26.09	**28.57**	27.95
	L	28.81	44.72	44.72	42.24	**47.82**	44.10
	M	23.52	33.77	33.28	32.42	**35.17**	34.72
BioASQ 5b	S	41.33	39.33	41.33	42.00	44.00	**46.00**
	L	56.67	52.67	55.33	54.67	56.67	**60.00**
	M	47.24	44.27	46.73	46.93	49.38	**51.64**
BioASQ 6b	S	24.22	33.54	**43.48**	41.61	40.37	42.86
	L	37.89	51.55	55.90	55.28	**57.77**	**57.77**
	M	27.84	40.88	48.11	47.02	47.48	**48.43**

Notes: BioASQ 4b/5b/6b datasets are designed for NLP question answering tasks from BioASQ leaderboard (http://participants-area.bioasq.org/). Strict Accuracy (S), Lenient Accuracy (L) and Mean Reciprocal Rank (M) scores on each dataset are reported. The best scores are in bold, and the second best scores are underlined. The test results are compared between three types of BERT model that learn from five different corpora. (Lee et al., 2019)

However, even within the same bioMedical domain **ClinicalBERT** believes that clinical narratives have differences in linguistic characteristics from both general text and non-clinical biomedical text used above (Alsentzer et al., 2019). They list out nearest neighbors for 3 sentinel words for each of 3 categories within clinical and bioBERT domains. ClinicalBERT appears to show greater cohesion within its domain than BioBERT, Table 2.

Table 2: Nearest neighbors for 3 sentinel words for each 3 categories (Alsentzer et al., 2019)

Model	Disease			Operations			Generic		
	Glucose	Seizure	Pneumonia	Transfer	Admitted	Discharge	Beach	Newspaper	Table
BioBERT	insulin	episode	vaccine	drainage	admission	admission	coast	news	tables
	exhaustion	appetite	infection	division	sinking	wave	rock	official	row
	dioxide	attack	plague	transplant	hospital	sight	reef	industry	dinner
Clinical	potassium	headache	consolidation	transferred	admission	disposition	shore	publication	scenario
	sodium	stroke	tuberculosis	admitted	transferred	transfer	ocean	organization	compilation
	sugar	agitation	infection	arrival	admit	transferred	land	publicity	technology

Their research focuses on specialized clinicalBERT models by training on two different specialized corpora of clinical notes and discharges summaries using two BERT models of $BERT_{BASE}$ and BioBERT for comparison. Their results show that additionally specialized BioBERT received better performance. Moreover, out of five given tasks they performed, the researchers argue that pure BioBERT only performs better on de-identification (de-ID) tasks because of its fundamental facets.

Table 3: Accuracy (MedNLI) and Extract F1 score (i2b2) across various clinical NLP tasks

Model	MedNLI	i2b2 2006	i2b2 2010	i2b2 2012	i2b2 2014
BERT	77.6%	93.9	83.5	75.9	92.8
BioBERT	80.8%	**94.8**	86.5	78.9	**93.0**
Clinical BERT	80.8%	91.5	86.4	78.5	92.6
Discharge Summary BERT	80.6%	91.9	86.4	78.4	92.8
Bio+Clinical BERT	**82.7%**	94.7	87.2	**78.9**	92.5
Bio+Discharge Summary BERT	**82.7%**	94.8	**87.8**	78.9	92.7

Notes: MedNLI is a natural language inference task (Romanov & Shivade, 2018) and i2b2 are named entity recognition (NER) tasks (*I2b2: Informatics for Integrating Biology & the Bedside*, n.d., p. 2; Sun et al., 2013, p. 2; Uzuner et al., 2011, p. 2)

Similarly, **BlueBERT** additionally trains the pre-trained BERT with their own version of corpora that includes PubMed (*PubMed*, n.d.) and MIMIC-III clinical documents (*MIMIC*, n.d.). However, their researcher came up with a benchmark that is similar to the General Language Understanding Evaluation benchmark or GLUE (*GLUE Benchmark*, n.d.) but for the biomedical domain. Biomedical Language Understanding Evaluation benchmark or BLUE (*Ncbi-Nlp/BLUE_Benchmark*, 2019/2021) consists of 5 tasks and 10 datasets that are specifically NLP tasks for the biomedical domain.

Table 4: BLUE benchmark datasets and tasks (Peng et al., 2019)

Corpus	Train	Dev	Test	Task	Metrics	Domain	Avg sent len
MedSTS, sentence pairs	675	75	318	Sentence similarity	Pearson	Clinical	25.8
BIOSSES, sentence pairs	64	16	20	Sentence similarity	Pearson	Biomedical	22.9
BC5CDR-disease, mentions	4182	4244	4424	NER	F1	Biomedical	22.3
BC5CDR-chemical, mentions	5203	5347	5385	NER	F1	Biomedical	22.3
ShARe/CLEFE, mentions	4628	1075	5195	NER	F1	Clinical	10.6
DDI, relations	2937	1004	979	Relation extraction	micro F1	Biomedical	41.7
ChemProt, relations	4154	2416	3458	Relation extraction	micro F1	Biomedical	34.3
i2b2 2010, relations	3110	11	6293	Relation extraction	F1	Clinical	24.8
HoC, documents	1108	157	315	Document classification	F1	Biomedical	25.3
MedNLI, pairs	11232	1395	1422	Inference	accuracy	Clinical	11.9

With their defined benchmark, they have performed a comparison between multiple state-of-the-art models such as ELMo, BioBERT and their own pretrain BERT. They show that their results are the best with slight improvement from BioBERT. However as they are the one who define this benchmark, this could possibly be due to some bias of their training dataset. Regardless, this Benchmark is useful for the development and experiment of this thesis.

Table 5: Baseline performance on BLUE benchmark task test sets (Peng et al., 2019)

Task	Metrics	SOTA*	ELMo	BioBERT	Our BERT			
					Base (P)	Base (P+M)	Large (P)	Large (P+M)
MedSTS	Pearson	83.6	68.6	84.5	84.5	**84.8**	84.6	83.2
BIOSSES	Pearson	84.8	60.2	82.7	89.3	**91.6**	86.3	75.1
BC5CDR-disease	F	84.1	83.9	85.9	**86.6**	85.4	82.9	83.8
BC5CDR-chemical	F	93.3	91.5	93.0	**93.5**	92.4	91.7	91.1
ShARe/CLEFE	F	70.0	75.6	72.8	75.4	**77.1**	72.7	74.4
DDI	F	72.9	78.9	78.8	78.1	79.4	**79.9**	76.3
ChemProt	F	64.1	66.6	71.3	72.5	69.2	**74.4**	65.1
i2b2	F	73.7	71.2	72.2	74.4	**76.4**	73.3	73.9
HoC	F	81.5	80.0	82.9	85.3	83.1	**87.3**	85.3
MedNLI	acc	73.5	71.4	80.5	82.2	**84.0**	81.5	83.8
Total			78.8	80.5	82.2	**82.3**	81.5	79.2

Notes: SOTA, state-of-the-art as of April 2019. BlueBERT has performed 4 different sizes and corpora of models for comparison.

However, these papers did not deeply study into the details of factors affecting each domain language. **BioMegatron** paper performs a more detailed analysis on subword vocabulary, model size, and domain transfer to evaluate impact in NLP tasks (Shin et al., 2020). They conclude that a language model performs best when it's targeted on domain and application. This also aligns with the "no free lunch" theorem in machine learning that there are no master models that can perform well for all tasks but good enough if we only targeted one.

2.3 Conclusion on Text representation

The process of performing additional training for fine-tune downstream tasks on pre-trained data-rich models is called **Transfer learning**. This technique has emerged as a powerful method in NLP because it gave rise to diversity of approaches, methodology and practice (Raffel et al., 2020). Transfer learning is an optimal way to build a model because it saves time by avoiding training from scratch and gets better performance as extension training converges better (CITED). Transfer learning enables us to develop skillful models with less data. This fit our scenarios of our BERT Language Model that was pre-trained on general corpora then transferred to biomedical corpora.

From an overview standpoint, our approach to build models for downstream NLP tasks has been changed. Instead of directly using input data such as sentences or documents to make classification, we use Language Model to capture semantic meaning and characteristics of those input and embedded into its data representation. Using this representation, we build another model such as a classifier on top to fine-tune for downstream NLP tasks, Figure 11. In order to maximize LM performance, transfer learning technique is used to learn from general corpora then shifted to domain specific corpora.

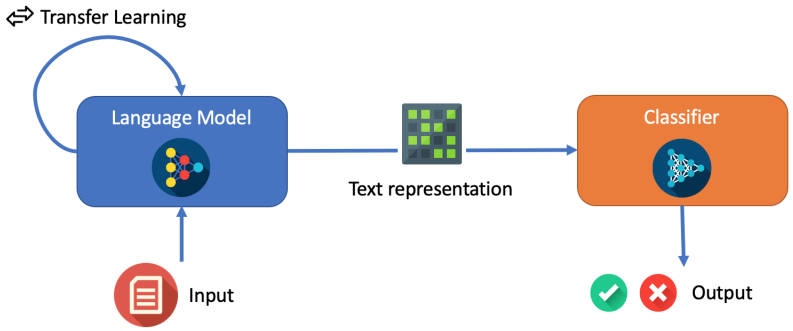


Figure 11: Overview of Modeling with Language model and Text representation

Identical to our thesis, the goal is a classification task given a radiology report to predict the presence of a fungal disease. We could simply use a Feed-forward Neural Network (FNN) with softmax to get our results, Figure 12. However, one of our constraints that we foresee is that we might not have a lot of labeled data for tasks. Semi-supervised (Zhu & Goldberg, 2009) and Active learning (Settles, 2012) might be our alternative solutions, allowing us to leverage the unlabelled data, for the next step of our research

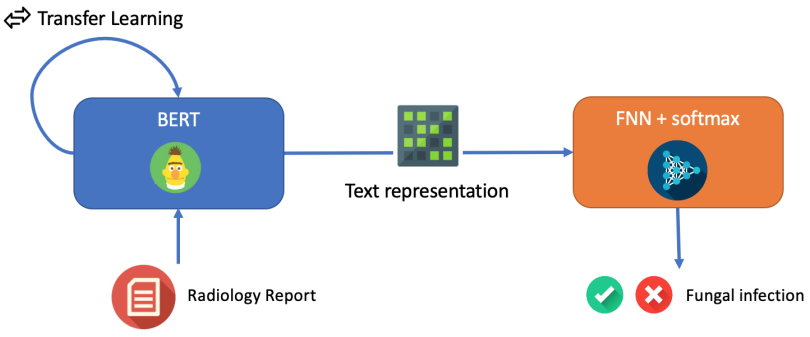


Figure 12: Overview of Modeling of Fungal infection disease detection

3. Summary of the State of the Art

The study of Natural Language Processing has been developed and improved over the past decade. Researchers in the field have built on top or come up with methodology to understand the semantic meaning of words. However, natural language is inherently complicated. There could be countless hidden meanings behind a single word. Humans learn to understand those by experience. Machines, unlike humans, could only learn based on the input we have given them directly, or external knowledge sources they have access to.

From our literature review, we have seen various attempts to extract those features. At every stage, researchers came up with a good text representation that could improve the state-of-the-art. However from those literatures, we can conclude that non-contextualize word representations such as Bag-of-words, Co-occurrence Matrix, and Word2Vec are no longer a good approach because of the sensitivity of meanings on each word. Contextualized word representation became more popular because it could capture those hidden meanings and dynamically adjust depending on our focus. The latest state-of-the-art BERT follows the same manner.

BERT or Bidirectional Encoder Representations from Transformers is a Language Model that takes input and converts it into text representation. It is a pre-trained Language Model that was learned from the large general corpora of BookCorpus and English Wikipedia. It looks at the whole input context that was given and performs transformations to embedded semantic meaning of each word in its given context. BERT's output is later used on downstream models to predict and classify NLP tasks.

However, as we stress the importance of words sensitivity towards its contexts, general BERT is not customized for a specific domain task that has its own corpora with distribution of words shifted from general ones. Transfer learning became an important element to guide the BERT model into a better performance. This was done by additional training the pre-trained BERT. There is multiple evidence of improvements when pre-trained BERT on biomedical corpora such as BioBERT, ClinicalBERT and BlueBERT. But because it is only as good as what it was trained for, our Radiology report customized version doesn't yet exist. Nevertheless, BERT has surpassed multiple state-of-the-art in both general and biomedical NLP domains. It has generated various promising outcomes throughout several NLP benchmark tasks. Considering all the elements mentioned above, we propose building a system that performs Text Classification using BERT with pre-trained radiology reports data. Our system should achieve a satisfactory outcome of detecting the presence of fungal disease. We hope to use our system to assist General Practice with their diagnosis to help every patient and save their lives.

4. Conclusion

In this paper, we have addressed the general problem of valuable data being overlooked. This problem has occurred and raised concerns in the biomedical domain as well, where various reports of physical examination on the patient have been documented. This information could be crucial as it could determine patients' health. We acknowledge this problem and remodel this challenge into NLP tasks of identifying the fungal infections based on radiology reports. In particular, we will be performing text classification using radiology reports as predictors to classify the possibility of fungal infections.

Our studies in literature have shown us that with recent NLP approaches it is possible to solve the challenge. We have gone through the history of text representation in NLP and improvement of the state-of-the-arts. The latest paper exhibits contextualizing word representation using bidirectional encoder representation from transformer (BERT). This method of Language Modeling allows us to embed semantic meaning of sentences into vectors dynamically. Thus, BERT that was pre-trained on general corporas require adjustment to demonstrate maximum performance when handling specialized domain corporas such as biomedical. Specifically, extended BERT needs to be trained on corporas of the focus because even within the same domain word distribution could be shifted. We acknowledge that limitation and will be focusing our thesis onto radiology report distribution of words. We believe that if our Language model could correctly be embedded with radiology report characteristics. Our downstream classification accuracy to predict the possibility of patients infected by fungal infection disease would be enhanced.

Part 2: The Research Paper

Fine-tuning Pretrained Language Models for Biomedical Tasks

Jirarote Jirasirikul
Monash University
jirarotej@gmail.com

Ehsan Shareghi
Monash University
ehsan.shareghi@monash.edu

Reza Haffari
Monash University
gholamreza.haffari@monash.edu

Abstract

Contextualized representation using pre-trained language models was able to encode high-quality semantics that offers good performance on downstream NLP tasks. The well-known BERT architecture achieved state-of-the-art results on various tasks and benchmarks. In the Biomedical domain, two prominent domain-specialised BERT variants are BioBERT and PubMedBERT that have achieved state-of-the-art results on Biomedical Language Understanding and Reasoning Benchmark leaderboard (BLURB). While successful, there are various ways to tailor these models for downstream tasks specifically. Our research explores further into the hallmark of cancer (HoC) and PubMedQA tasks to verify the quality of representation from these models. We fine-tune these language models via various strategies such as input preprocessing, ensembling, and adapter layers. The empirical results substantiate the necessity of using domain-specific language models when dealing with special domains, and show that fine-tuning the underlying language models can generate more vigorous outcomes with a trade-off. The ensembling technique builds a more robust model for dealing with precision in exchange for recall. The adapter adds downstream tuning parameters in between layers of the language model. This allows the parameter's weight to be more effective, however, GPU is became mandatory for fine-tuning downstream tasks together with the language model. Moreover, our observations point out that, in contrast with general domain, in special domains including more context could confuse models during learning and prediction.

1 Introduction

Pre-trained language models have become a standard approach on handling textual dataset in natural language processing (NLP) applications. The contextualized representations that build from these models encodes context-dependent semantics which thumps the past methodology. The important foundation of this approach is to let the language model learn via a self-supervised learning objective function (e.g., masked language modelling) on large corpora. Once pre-trained, language models can be shared and utilized on other NLP tasks via transfer learning provided some additional fine-tuning on top.

One of the state-of-the-art language model architectures is Bidirectional Encoder Representations from Transformers (BERT) that build upon an attention-based mechanism, which enables the model to capture sequential information from the input (Devlin et al., 2019). BERT uses masked language model (MLM) loss to acquire both left and right context semantics of the word. The basic model, known as BERT-base, self-supervised on general domain corpora, i.e. wikipedia and web crawling, has yielded the state-of-the-art performance on the General Language Understanding Evaluation (GLUE) benchmark (Wang et al., 2018). Later researchers built upon the same architecture with different sets of corpora creating multiple variants of BERT models that can outperform in particular domain tasks.

The Biomedical domain is considered to have a high impact because of this uniqueness in word distribution. This domain is regarded as challenging due to data being sensitive and technical terminologies. The domain has its own NLP benchmark called the Biomedical Language Understanding and Reasoning Benchmark (BLURB) (Gu et al., 2021). Two prominent domain-specialized BERT variants in the BLURB leaderboard are: (1) BioBERT that is continually pre-trained from BERT-base on PubMed abstract corpora (Lee et al., 2019), and (2) PubMedBERT that pre-trained on PubMed abstract corpora from scratch (Gu et al., 2021). In this paper, we observe the quality of representations from a set of mentioned pre-trained language models on two complex biomedical tasks: (1) classification Hallmark of cancer (HoC) task, and (2) question-answering PubMedQA task.

Our research aims to explore different ways to fine-tune the underlying BERT models for downstream tasks. We implement three types of strategies for fine-tuning. These consist of (1) preprocessing the task input data according to characteristics to highlight its features, (2) building an ensemble model to enhance the robustness of the prediction, and (3) utilizing adapter layers architecture (Houlsby et al., 2019) in BERT to minimize the number of tuning parameters without much task compromise compared to fine-tuning a full model (e.g. underlying BERT as well as the classification layer). We perform

minimal changes between settings to evaluate influence separately.

We have evaluated the empirical performance of all strategies. A general observation on the result substantiates the essential of using biomedical domain-specific language models when dealing with biomedical domain tasks. The results show that the BioBERT and the PubMedBERT outperform the general domain language model BERT-base with significance. Furthermore, the domain-specific approach requires much less fine-tuning to reach the optimal performance. This outcome supplements other research that selects the same domain specific corpora between a language model and a task. Moreover, we address the critical limitation of current BERT that can handle a maximum of 512 tokens when dealing with paragraph context such as paper abstract.

As we explore deep into tasks, we realize that the biomedical domain is delicate and should consider other performance evaluation metrics aside from accuracy or f1 score that was used on the leaderboard. For example, false negatives are more acceptable than false positives when identifying disease. In ensemble technique setup, we consider precision and recall to capture these additional aspects. Our ensemble models have demonstrated that they can improve accuracy/f1 score on the leaderboard in exchange of precision or recall that can capture type I/II error better. In the biomedical domain, these models might still be useful depending on the objectives agreed upon. In the adapter layers architecture setup, we notice compulsory requirements on the computer processing unit that needs high GPU access for fine-tuning. The adapter model has task-specific weight parameters ingrained within the adapter layer that is structured in between the language model layer. Even though the language model parameter is frozen, each epoch of fine-tuning will create a new set of representations for calculating loss. Thus, this approach cannot cache the representation, as the language model is not distinct from the downstream model. However, because the fine-tuning parameters are embedded in the language model, their weights are more significant to the representation. This yields an on-par or greater prediction performance on some aspects.

In conclusion, we have compared sets of general and specific domain language models on two biomedical NLP tasks. We observe that using a biomedical domain-specific language model could generate a good quality representation better suited for the biomedical tasks. However, selecting the best domain-specific model is still inconclusive. We could improve these models using ensembling or adapter layers but there would be some downside in exchange. This trade-off may be acceptable and still useful especially in the dedicated domain of biomedical. We hope that our research will inspire others to see the benefit of task-oriented models. We would like to acknowledge huggingface.co and adapterhub.ml for contribution toward the NLP community and the source code of this paper experiment setup. The code of our experiment are available at:

https://github.com/blizrys/BioMed-BERT-Eval

2. Related work and background

The use of language representations to capture textual semantic meaning goes way back in history. The machine needs this conversion as it could only process numerical values. However, a simple transformation cannot encode a substantial meaning. This provoked representation techniques to evolve in order to catch up with the rapid growth of data and solving various NLP tasks (*Evolution of Natural Language Processing*, 2020). These representations are later used as an input to most of the downstream tasks.

One well-known form of representation that embedded some semantic meaning of each word is Word2Vec. This method was introduced in late 2013 (Mikolov et al., 2013), which generates a vector representation for a word. It is constructed by going through a large-scale corpus considering a window of words around a target word to predict it correctly . It is a simple and easily accessible approach especially to learn word embeddings. Word2Vec's major downfall is that it gives a static transformation to the word given. In other words, its embeddings are context-independent. As a result, it cannot handle words that have multiple meanings (which could only be resolved given their context). There are attempts to improve this methodology via other embedding learning techniques, namely GloVe and FastText (Joulin et al., 2016; Pennington et al., 2014). However, these methods cannot overcome the aforementioned problem. Nevertheless, Word2Vec remains as a very popular text representation learning technique due to its simplicity. The Word2Vec contributed a lot to the NLP field as a foundation to text representations.

Context-dependent representations have been invented and have become popular in recent years. They allow the meaning of words to change dynamically according to the input context. Early adoption of this notion is an Embeddings from Language Model or ELMo (Peters et al., 2018). It was presented as a method to derive word representation vectors from the context using Long-short term memory (LSTM) recurrent neural networks. It was trained bi-directional on a large text corpus to be able to get word representation that can cover both pre- and post-sentences of the focus word. Later in 2017, The transformer paper "attention is all you need" was published (Vaswani et al., 2017). This comes with the fact that the

attention mechanism deals with long-term dependencies better than LSTM. Researchers combined these ideas together and constructed Bidirectional Encoder Representations from Transformers known as BERT. This method can encode semantic meaning as a context-dependent representation while using a transformation mechanism. This approach has become the latest state-of-the-art form of textual representation (Devlin et al., 2019).

A technique of pre-training these models in a rich environment full of various corpora before utilizing it in other tasks is called transfer learning. This technique is widely adopted nowadays as it lessens effort when applying it to downstream tasks. The researcher can share models that have been pre-trained in a specific setting with others (*Hugging Face – The AI Community Building the Future.*, n.d.). However, choosing appropriate pre-trained settings could be challenging. As we stress the importance of the word sensitivity towards its contexts, it is necessary to pre-trained the language model according to its downstream domain tasks. Each corpus could have an essential shift in word distributions. Generally speaking, the dictionary built from each corpus is not the same. Original BERT was not customized for a specific domain. It was trained on Wikipedia and Books, thus, we called this a general domain BERT.

A couple of publications have focused on selection of specific domain corpora for pretraining. In the biomedical domain, two prominent BERT models that surpass original BERT when handling Biomedical NLP tasks are BioBERT and PubMedBERT (Gu et al., 2021; Lee et al., 2019). These two language models are similar in terms of pre-training on specific domain corpora. However, the major difference is that BioBERT uses an approach to extend an existing pretrained BERT from the general domain by performing continual training on Biomedical corpora. In contrast, PubMedBERT trained its model from scratch on Biomedical corpora. Both approaches have demonstrated an improvement of results in Biomedical NLP benchmark tasks, known as BLURB (*BLURB Leaderboard*, n.d.). It generates a representation that is embedded with biomedical semantics before being fine-tuned in each task. They show great signs toward a better way of capturing textural semantics in a specific domain. As these approaches are quite recent, they are pending exploration on their maximum capability.

Fine-tuning language models is effective in improving downstream task performance. The current approach is considered parameter inefficient as an entire new downstream model is required for every task (e.g., fine-tuning the full model along with a classification head). One suggested parameter-efficient arrangement is to add adapter modules in between layers of BERT architecture (Houlsby et al., 2019). Adapters allow the underlying BERT model to be frozen while only the added adapters are trained on task data to do the adaptation of the BERT model for the task. By fixing the original network parameters and fine-tuned parameters in the adapter modules, they attained near state-of-the-art performance on GLUE general domain NLP benchmark tasks (Wang et al., 2018).

3. Methodology

3.1 Overview

In this section, we provide a brief explanation of using neural language models to perform NLP tasks in Figure 1. This research focuses on the state-of-the-art neural language model Bidirectional Encoder Representations from Transformers (BERT). The definition of each component is listed below:

3.1.1 Input. The input is a text span that consists of a sentence or multiple sentences. It could be a single sentence, a combination of multiple sentences, or a combination of questions and answers in a document. Essentially, the input will be tokenized using BERT tokenizer, which performs (1) WordPiece algorithm to generate tokens, (2) inserts [CLS] token at the beginning of context, and (3) separator between sentences by a special token [SEP]. It is limited to a maximum size of 512 tokens due to the maximum sequence length of BERT. The input may preserve the case (cased) or convert all characters to lowercase (uncased) according to the settings.

3.1.2 Language Model. The model introduces a multi-layer and a multi-head self-attention mechanism. We used the pre-trained weight of multiple BERT models from Huggingface.co as our base model to transform input data into a text representation.

3.1.3 Text representation. An output from the BERT language model in a form vector for each input token. For n-tokens sentence, an output of n × 768 dimensional vectors are create. Each dimension carries a weight that holds context-dependent semantics. We use the first position vector, associated with the [CLS] token, as the representation for downstream classification. This is because the BERT objective seemingly encapsulated a sentence-wide into the first position output.

3.1.4 Downstream Model. An overlay model on top of the language model that corresponds to a selected NLP task objective. The model takes text representation as input and generates predictions. Our focus is to aim deep into Biomedical benchmark tasks given in BLURB leaderboard. The Hall-of-Cancer (HoC) and the PubMedQA tasks have been selected as our focal point in biomedical NLP tasks.

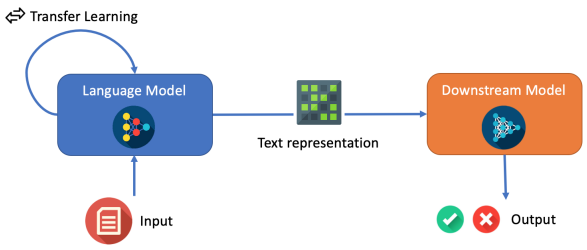


Figure 1: The figure shows an overall architecture of an NLP prediction task from start-to-end. The language model transforms text input into its representation. The downstream model takes language representation and processes task prediction. In our research, we use the BERT language model that was pre-trained on a specific domain, which transfers its learning from a large corpus onto our input.

3.2 Dataset Preprocessing

In this research, we noticed that previous works have been focusing on generating better results using a new type of language model without understanding dataset behavior, thus, they use the input dataset as what is given to them. Due to that reason, we believe that each dataset has more to offer as it could potentially have its characteristics such as number of sentences and labels, or dependency between sentences. Our approach is to understand the dataset from a statistical and practical point of view. We will preprocess the data before feeding it into our corresponding model for evaluating the performance. The setup will focus on recognizing the impact of the input. Generally from Figure 1, we can see that our inputs will need to be processed with a language model before going through the downstream model. This means that if our input is different even with slight modification, text representation, which was the output of the language model, could have very different weights. It also means that the format of our input that we perform text-preprocessing will have consequences on our final prediction output. We freeze the language model and fine-tuned the downstream task layer according to the transformed input. In this setup, the preprocessing action will be defined according to the statistical characteristic of the dataset. The action will be focusing to heighten the dataset feature such as adding more context, or prioritize important abstract.

3.3 Ensemble Model

This section aims to utilize previous models with an ensemble approach to generate better and more robust models. The ensemble method is a machine learning technique that combines several base models to produce one optimal predictive model. The results of an ensemble model are usually a function of the average or a majority vote of those multiple base models. We set up this ensemble model by generating 10 fine-tuned model

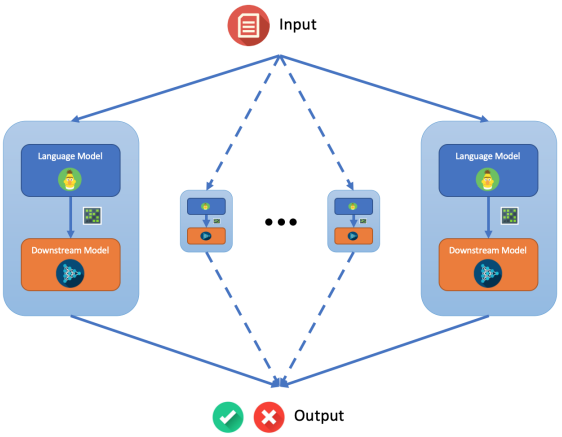


Figure 2: An ensemble model architecture to calculate final downstream output. Same input will be used in multiple models that were initialized differently to train and predict test value as a result. The result of each model will be aggregated using an ensemble technique to get the final prediction.

checkpoints from the previous experiment using the best configuration but different random initialization of the neural network prior to training. These models are a mixture of BioBERT and PubMedBERT language models. We expect that this combination will take the best semantic interpretation of both language models. We take these selected models and ensemble the prediction results with an ensemble function. We select the proper function by analyzing the results that will improve model prediction in a certain way. We summarize our findings to see the potential of utilizing the ensemble technique on Biomedical domain tasks.

3.4 BERT Adapter

BERT Adapter is a new lightweight architecture as an alternative to full fine-tuning of a pre-trained language model on a downstream task. It consists of adding only a small set of newly introduced parameters called adapter modules in between the transformer layers. As a result, instead of the necessity to fully fine-tune the new set of parameters on the top layer of the language model for the downstream task, this architecture allows us to fine-tune only a small number of parameters. It is shown that this approach could achieve results comparable to the traditional approach while being more parameter-efficient, faster during training, and more compact to be shared. We instantiate adapter-based tuning models using the proposed architecture, following https://adapterhub.ml/ guides, and compare them against the traditional top layer fine-tuning approach on Biomedical NLP tasks. The original comparison was done on the General domain, which does not have a complex word distribution.

Statistic	Train	Dev	Test
Dataset size			
Documents	1295	186	371
Sentences	12119	1798	3547
Avg. sentences per document	9.36	9.67	9.56
Prop. of cancers (%)			
activating invasion and metastasis	15.52	16.71	15.05
avoiding immune destruction	6.64	3.77	4.30
cellular energetics	5.25	5.12	9.68
enabling replicative immortality	6.72	6.47	2.15
evading growth suppressors	13.05	14.29	9.68
genomic instability and mutation	17.14	17.52	24.73
inducing angiogenesis	7.80	6.20	10.22
resisting cell death	23.86	21.29	22.04
sustaining proliferative signaling	24.32	28.30	22.58
tumor promoting inflammation	13.90	10.78	10.75

Table 1: Hall of Cancer dataset statistics.

Default setting		Previous n-sentences setting	
Label	Sentence	n = 1	n = 2
X	A	A	A
Y	B	A + B	A + B
Z	C	B + C	A + B + C

Table 2: The table shows how sentence(s) are formatted as context before using it as an input in the "previous n-sentences" settings.

Statistic	Train	Dev	Test
Dataset size	450	50	500
Prop of yes (%)	55.20	55.20	55.20
Prop of no (%)	33.80	33.80	33.80
Prop of maybe (%)	11.00	11.00	11.00

Table 3: PubMedQA dataset statistics. The values in the Train and Dev columns are average across all folds.

4. Experiments and Results

In this section we report our experiments on two NLP tasks. Tasks are imported from BLURB leaderboard to perform prediction. We analyse the results and summarize model behaviour.

4.1 Datasets & Tasks

4.1.1 Hallmarks of Cancer (HoC)

The dataset consists of 1852 PubMed publication abstract documents. Each sentence of the document was manually labeled by an expert in taxonomy. Labels can consist of 10 types of cancer that came from 37 classes in a hierarchy. Zero or more labels can be assigned to each sentence at the sentence-level. For evaluation, labels from the same document have been aggregated together, whereby a document that has at least one sentence identifying that cancer type will also be considered for the document. We called this a document-level label. The statistical proportion of labels in the dataset is shown in Table 1. The macro-average F1 score is used to evaluate all cancer types' predictions.

As the evaluation is being conducted at a document level, this means that there is a gap between sentence level and document level during prediction. The final results ignored the prediction accuracy of each sentence. It focuses entirely on some sentences in the document that could potentially identify cancer types. Our assumption is from knowing the BERT capability in creating semantic meaning from multiple sentences. We decided to try concatenating n sentences within the document together. This setup let us observe whether BERT could handle a better understanding of the overview document given more context. We called this "Previous n-sentences" setup in Table 2. Moreover, we analyze the results to observe complicated cancer types that are harder to predict than others. To sum up, we form two following sub-experiments as Dataset preprocessing: (1) to see the influence of adding more context when generating language representation by including n-previous sentences, and (2) Identifying the challenging types of cancer that are tough to predict based on their corresponding prediction performance. In addition, we take the best setup of this experiment to our next experiments with ensembling and BERT adapter models, mentioned in the methodology.

4.1.2 PubMedQA

The task is to answer research questions with multiclass labels given a corresponding context. Each instance consists of (1) a question for an existing research article, (2) abstracts corresponding to the question without conclusion, (3) a long answer that concludes the abstracts and presumably answers to the instance question, and (4) a label of yes/no/maybe that summarizes the conclusion. PubMedQA has been manually labeled by experts with two setups: context-free or context-required, whereby context-free considers the question and long answer sentences and context-required considers the question and abstracts. The final label is a consensus between these two approaches. The statistical proportion of labels in the dataset is shown in Table 3. The accuracy scores are calculated as macro-average across 10 folds from the model prediction against the final label using a context-required setup for the model.

As the context is formed from a combination of the question and the abstracts, humans can simply identify which is which, however, the machine only learns from its representation. The priority of these sentences is vital to how the representation is assembled. Ideally, we might select only important abstract sentences to be included in the context, though expertise would be required. Our experiment will be to formulate the settings between

Model	BLURB Score	HoC	PubMedQA
PubMedBERT (uncased)	81.50	82.62	55.84*
BioBERT (cased)	80.34	81.54	60.24*
BERT base (uncased)	76.11	80.20	51.62*

Table 4: The table shows an up-to-date baseline score that was published on the BLURB leaderboard. BLURB Score is an overall score across all biomedical benchmark NLP tasks. HoC and PubMedQA are tasks in the benchmark. Remark(*): For PubMedQA, we used the score from the publication instead because the leaderboard displays a macro-average score between all Question Answering tasks.

putting the question before or after abstracts. This will let us see whether the prioritization of QA is crucial to building quality semantic representation. In addition, we will also look into the context-free and context-required format that was dropped out from the BLURB paper but exists in the original PubMedQA paper. To sum up, we form two following sub-experiment: (1) considering the position of question and abstracts, and (2) demonstrate the importance of long-answer towards its prediction by comparing context-free and context-required setups. Similarly, after retrieving the best setup hyperparameter, we perform additional evaluations of the ensembling and the BERT adapter models.

4.2 Model Configuration

Our experiment uses google colab pro with Tesla P100 GPU 16GB RAM. In each setting, we tried to change an optimal configuration for our results to be interpretable. Our initial BERT model weights for the original BERT, BioBERT, and PubMedBERT language models are being loaded from an open-source https://huggingface.co/. An up-to-date score of these models on the BLURB benchmark is presented in Table 4.

For fine-tuning, we use Adam as an optimizer and tune the hyperparameter with combinations of learning rate (1e-5, 3e-3, 5e-5), batch size (16, 32), and use maximum BERT token size of 512. This is an identical setup to the original paper for the BLURB benchmark. The downstream models and evaluation metrics are different according to the NLP tasks in Table 5.

On the other hand, we compare the BERT Adapter model setup on biomedical NLP tasks by following the adapter architecture setup. Due to computation limitations found during the experimental period, we use a provided huggingface's trainer framework with a fixed batch size of 8 and a maximum BERT token size of 512. We keep the learning rate (1e-5, 3e-3, 5e-5) to be as similar to the previous experiment as possible. We rebuild original architecture and adapter architecture to compare the results and observe the changes.

	Downstream Model	Loss Function	Evaluate Metric
HoC	Linear Layer	Binary Cross Entropy	Macro-F1
PubmedQA	MulticlassClassification Layer	Cross Entropy	Accuracy

Table 5: The table shows setup methodology according to each datasets.

Hallmark of Cancer			Previous n-sentences (F1 score)			
batch size	**lr.**	**Language Model**	**n = 0**	**n = 1**	**n = 2**	**n = 3**
16	5e-5	PubMedBERT (uncased)	**81.26**	79.54	79.11	77.93
		BioBERT (cased)	**82.33**	79.70	77.37	77.97
		BERT base (uncased)	**71.44**	67.32	67.55	65.72
32	5e-5	PubMedBERT (uncased)	**80.87**	79.30	79.32	77.90
		BioBERT (cased)	**82.47**	79.40	78.09	77.12
		BERT base (uncased)	**70.61**	66.28	66.87	64.75

Table 6: Evaluation results of different previous n sentences settings. We perform evaluation on two different batch sizes while keeping the same learning rate. Bold text shows the best F1 score for each model configuration.

4.3. Results

In this section, we conduct multiple comparisons to evaluate and analyze each setting mentioned above. We follow our methodology setting one by one and explain our findings within each subsection.

4.3.1 Input Preprocessing

This setting adjusts input data that is being processed and controls the language model. The underlying BERT model is frozen and the classifier layer is fine-tuned according to the data representation. Each dataset has a different setup that will be evaluated separately to observe the prediction performance that was impacted after representation has been altered.

Hall of Cancer

For HoC datasets, our assumption is based on the BERT's ability to build semantic representation from multiple sentences, in this case at a document-level. We conduct different hyperparameter tuning, given each sentence has its own corresponding label in Table 6. We discover that adding multiple sentences into one representation yields worse prediction results. From the results in a previous n-sentences setup within the same hyperparameter set, the best performance is demonstrated when n=0. Notably, results are very consistent that adding more sentences will lead to the lower performance in prediction, regardless of language model used.

To understand the cause of the behaviour, we assess the average number of tokens in a sentence of the HoC dataset. Our inspection reveals that each sentence consists of approximately 50 tokens. Considering our largest experiment when n=3, the number of tokens of concatenate sentences would be approximately 200 tokens. This is far from BERT maximum tokens limitation that can process

HoC	Label count			PubMedBERT (uncased)			BioBERT (uncased)		
batch size = 32, lr = 5e-5	Train	Dev	Test	Pres.	Recall	F1score	Pres.	Recall	F1score
activating invasion and metastasis	448	64	62	86.79	74.19	80.00	86.27	70.97	77.88
avoiding immune destruction	185	20	14	**45.00**	64.29	**52.94**	**50.00**	50.00	**50.00**
cellular energetics	136	44	19	86.67	68.42	76.47	93.33	73.68	82.35
enabling replicative immortality	219	11	24	85.71	75.00	80.00	82.61	79.17	80.85
evading growth suppressors	268	23	53	68.57	**45.28**	54.55	68.75	**41.51**	51.76
genomic instability and mutation	523	103	65	72.73	61.54	66.67	75.00	60.00	66.67
inducing angiogenesis	238	60	23	85.71	78.26	81.82	69.57	69.57	69.57
resisting cell death	602	72	79	87.10	68.35	76.60	84.06	73.42	78.38
sustaining proliferative signaling	674	85	105	70.51	52.38	60.11	76.32	55.24	64.09
tumor promoting inflammation	375	45	40	74.07	50.00	59.70	76.67	57.50	65.71

Table 7: Prediction evaluation results for HoC dataset in detail. Each row shows the type of cancer label and evaluation score corresponding to it. Bold and underline text shows the first and second lowest score of each model, which determines the hard classification type of cancer.

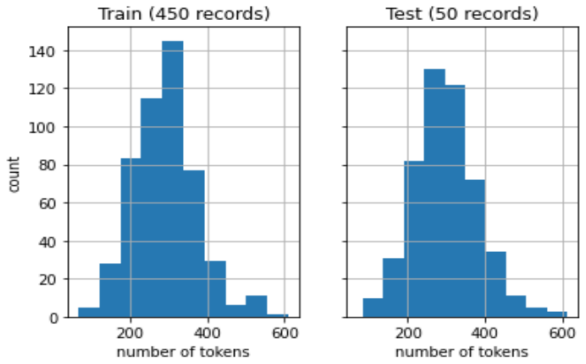


Figure 3: Histogram of the number of tokens in PubMedQA

upto 512 tokens. Subsequently, we hypothesize that since BERT output representations are bound by 768 dimension-vectors, adding more sentences into each representation will cause each representation to lose its key meaning. We create a visualization in Figure 4 showing the weight of attention of the BERT model. There is evidence that the weight in the final attention layer is being swayed when the number of tokens in each representation increases.

In another aspect, we select one of the models above with a single sentence per label (n = 0) to observe prediction accuracy at the sentence level. The results are shown in Table 7 that avoiding immune destruction (IM) cancer type and evading growth suppressors (GS) cancer type yield a worse prediction across all types of cancers. Although it does show some correlation with the number of training instances, we believe the Ensemble model could improve this incorrect prediction. Furthermore, we will also concentrate on the Recall score. This is because false negatives when identifying cancer might be better than letting patients with cancer unidentified in medical domain modeling.

PubMedQA

For the PubMedQA dataset, our assumptions are based on the structure of the input context that is being fed into the model. BERT can generate language representation that

PubMedQA - 10 Fold			Classification Accuracy		
batch size	lr.	Language Model	BLURB	Q. + Abs.	Abs. + Q.
16	1e-5	PubMedBERT (uncased)	55.84	**51.72**	49.32
	3e-3			**52.42**	48.80
	5e-5			**51.90**	48.10
32	1e-5			**49.28**	44.58
	3e-3			**55.06**	52.92
	5e-5			**51.36**	46.84
16	1e-5	BioBERT (cased)	60.24	**50.10**	47.74
	3e-3			**51.46**	50.44
	5e-5			**49.58**	47.90
32	1e-5			**49.02**	44.88
	3e-3			**54.16**	53.32
	5e-5			**49.30**	47.56
16	1e-5	BERT base (uncased)	51.62	47.64	**49.10**
	3e-3			**48.46**	48.06
	5e-5			46.62	**47.84**
32	1e-5			43.00	**43.66**
	3e-3			47.62	**50.06**
	5e-5			**46.34**	45.60

Table 8: Evaluation results of different settings on PubMedQA dataset comparing input format of "Question before Abstracts" and "Abstracts before Questions". Bold text shows the highest accuracy in each setting model.

encodes semantic information for the whole context, however, we explore if the priority of sentence within the context matters. Our results in Table 8, show that putting a question before abstracts is consistent with getting better predictions results with biomedical language models. For the general domain BERT, the pattern is unclear. To understand the root cause of this behavior, we calculate the number of tokens of input, which has an average of 292 tokens per context, however, a portion of 1.55% of the training dataset and 16% of the test dataset could reach over 512 tokens per context, shown in Figure 3. We realize that this is beyond the capability of the current BERT model, which can take up to a limit of 512 tokens. This means regardless of which format of the input structure we used, some of the information was disregarded during the transformation. This finding is crucial as it has not been mentioned in previous papers. We believe that this finding might be the cause of our poor results in the "abstracts before question" setting, as some parts of the question might be cut off. It is very important to have a full question embedded in the language representation as the question determines the expected answer. Unfortunately, even if we realize this problem, we cannot improve the question before abstract setting with the current BERT limitation. This would require a selective method to identify important parts of the abstract to be included which is not simple and need expertise. Moreover, when considering the low score in the overall prediction of PubMedQA, It might also be due to the same reason as more sentences swaying the representation away. This opens up a future opportunity to research how to select necessary sentences before building representation.

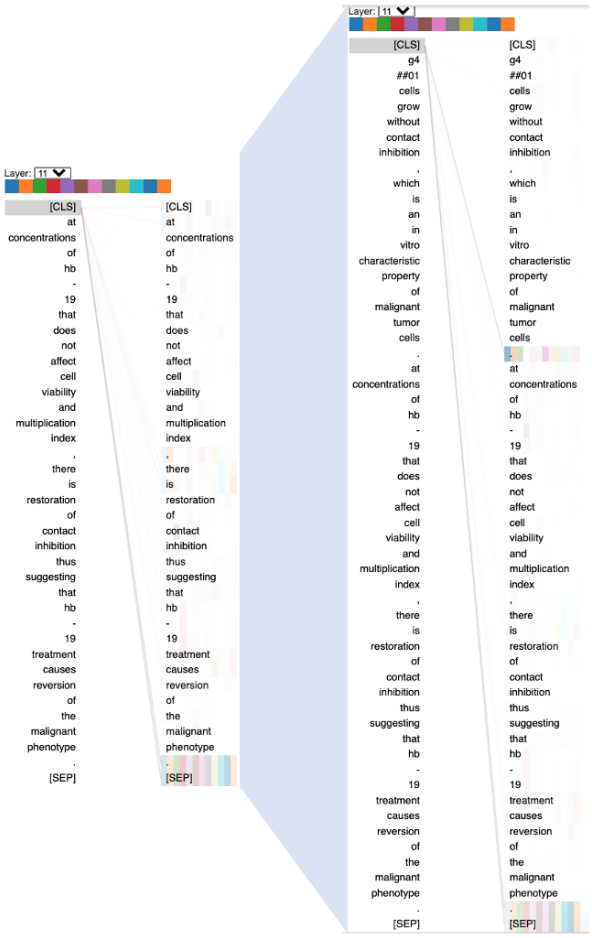


Figure 4: Visualization of BERT attention weight on the last layer. Left side is the weight of the CLS token in a previous n=0 setting (or single sentence setting). Right side is the weight of the CLS token in a previous n=1 setting (or two sentences setting).

In addition, we compare the missing evaluation in the BLURB leaderboard of using context-free and context-required settings in Table 9. This shows that context-free setting, where language representation was generated by including expert summarization (or long answer) of the abstract, is performing better than context-required setting. This is consistent with previous observation that the input that gives the best prediction are usually formed from short, concise, and contain key-idea of the things we need to predict.

4.3.2 Ensemble Model

In this subsection, we try to improve previous results by creating an ensemble model from PubMedBERT and BioBERT. We generate multiple models and use the ensembling function to aggregate their results.

Hall of Cancer

Results are shown in Table 10. For the 'Majority Vote' technique, although it improves the precision of our results, which ensures that our model does not falsely identify cancer, it yields a worse F1 Score overall. This

PubMedQA - 10 Fold			Classification Accuracy		
batch size	**lr.**	**Language Model**	**BLURB**	**Cont. req.**	**Cont. free**
16	1e-5	PubMedBERT (uncased)	55.84	51.64	**60.52**
	3e-3			51.56	**59.16**
	5e-5			51.92	**62.86**
32	1e-5			47.00	**57.14**
	3e-3			54.76	**64.40**
	5e-5			50.38	**60.16**
16	1e-5	BioBERT (cased)	60.24	49.38	**59.64**
	3e-3			50.60	**60.34**
	5e-5			51.18	**61.66**
32	1e-5			48.14	**57.84**
	3e-3			54.68	**66.14**
	5e-5			49.22	**61.30**
16	1e-5	BERT base (uncased)	51.62	47.68	**56.70**
	3e-3			48.42	**54.38**
	5e-5			46.60	**56.12**
32	1e-5			45.62	**53.92**
	3e-3			47.96	**60.98**
	5e-5			45.58	**56.82**

Table 9: Evaluation results of different settings on PubMedQA dataset on the missing setup in the original paper of Context-free and Context-required. Bold text shows the highest accuracy in each setting model.

Mix BioBERT and PubMedBERT	Label count			Majority Vote			Atleast 1 model		
batch size = 32, lr = 5e-5	**Train**	**Dev**	**Test**	**Pres.**	**Recall**	**F1score**	**Pres.**	**Recall**	**F1score**
activating invasion and metastasis	448	64	62	91.89	54.84	68.69	83.87	83.87	83.87
avoiding immune destruction	185	20	14	**58.33**	50.00	**53.85**	**42.86**	64.29	**51.43**
cellular energetics	136	44	19	90.91	52.63	66.67	88.24	78.95	83.33
enabling replicative immortality	219	11	24	100.00	54.17	70.27	76.92	83.33	80.00
evading growth suppressors	268	23	53	71.43	**18.87**	29.85	65.22	**56.60**	60.61
genomic instability and mutation	523	103	65	80.00	36.92	50.53	68.57	73.85	71.11
inducing angiogenesis	238	60	23	88.24	65.22	75.00	67.86	82.61	74.51
resisting cell death	602	72	79	95.35	51.90	67.21	79.01	81.01	80.00
sustaining proliferative signaling	674	85	105	76.60	34.29	47.37	71.28	63.81	67.34
tumor promoting inflammation	375	45	40	80.00	30.00	43.64	71.79	70.00	70.89

Table 10: Prediction evaluation results for HoC dataset in detail. We ensemble 10 different initialized models of BioBERT and PubMedBERT weight (5 each) and aggregate prediction results before evaluation. Each row shows the type of cancer label and evaluation score corresponding to it. Bold and underline text shows the first and second lowest score of each model, which determines the hard classification type of cancer.

would not be a preferable outcome in comparison with the leaderboard. However, this model might be used in real-life in which we need high precision to conclude if the patient has cancer or not. Meanwhile, the 'At least one model' technique gains a higher recall and also gives a promising F1 Score that outperforms most of the previous section's predictions. It might not be a good technique as this approach will keep increasing false negatives when more models are added. Using this technique is still a good approach to gain some indication of whether a patient has a chance of cancer before further diagnosis with another approach.

PubMedQA

We generate three sets of models (1) all models using PubMedBERT language model, (2) all models using BioBERT language model, and (3) a mix between PubMedBERT and BioBERT language models. Results are

PubMedQA	batch size = 32, lr = 3e-3	Weight Average		
Language Model	Ensemble Model	Precision	Recall	F1 score
All PubMedBERT	No ensemble (previous score)	56.56	59.20	57.12
	10 models	**56.55**	**59.80**	**57.57**
	100 models	55.68	59.00	56.57
	1000 models	55.48	58.40	56.14
All BioBERT	No ensemble (previous score)	50.04	52.20	50.94
	10 models	53.11	56.20	54.21
	100 models	53.26	57.00	54.59
	1000 models	53.60	57.00	54.66
Mix PubMedBERT and BioBERT	10 models	56.36	59.60	56.88
	100 models	55.66	59.40	55.93
	1000 models	55.59	59.60	56.13

Table 11: Prediction evaluation results for PubMedQA dataset. Comparing ensemble architectures between different numbers of models and different combinations of language models. Bold and underline text shows the first and second highest score of each metric. Weighted Average Recall score is equal to Accuracy.

shown in Table 11. Our analysis uses only the 'Majority Vote' technique to aggregate results as we only focus on accuracy. The results showed that ensemble models generally increase accuracy (or weighted average recall) scores. Results also show that the scores are more consistent toward multiple models. This shows that using an ensemble model will yield a more robust accurate score than using a single model prediction. Additionally, in a single language model setting, the representation could be dependent on which language model was used. However, when we mix these language models the results only show a minimal drop in the score while maintaining overall performance. This proves that ensembling is a powerful approach to construct a model.

4.3.3 BERT Adapter

In our final experiment, we evaluate the new architecture of the BERT with adapters. This setup allows us to compare between a traditional fine-tuned downstream model, where we freeze pre-trained BERT and train classification layer, and a fine-tuned adapter layer that is within the BERT language model. [1]

Hall of Cancer

We train HoC datasets on three types of BERT with both architecture and compare the results in Table 12. We noticed unusual behavior from the observation. In the HoC, if the learning rate becomes large in small batch sizes, the model prediction is equal to the null model, where prediction of all cancer is equal to 0 (no cancer). We can see this happens when the learning rate is at 3e-3. This behavior is not obvious in the large batch sizes that we have performed in the previous section, although we can see a slight decrease in performance. Despite that behavior, in our findings, we recognize that training BERT adapters

[1] Google colab pro processor unit cannot handle large batch size with maximum 512 tokens when training the language model. As adapter architecture needs to fine-tune downstream parameters weight within the language model, this causes a memory outage. We decided to lower down the batch size to a size of 8 to continue the research. We keep other values in the hyperparameter set similar to make the results comparable for interpretation.

HoC 6 epoch			Classification F1 Score		
batch size	lr.	Language Model	BLURB	Tradition	Adapter
8	1e-5	PubMedBERT (uncased)	82.62	**85.93**	83.04
	3e-3			53.68	53.68
	5e-5			56.28	**88.64**
8	1e-5	BioBERT (cased)	81.54	**87.14**	82.95
	3e-3			53.68	53.68
	5e-5			61.23	**88.84**
8	1e-5	BERT (uncased)	80.20	**84.55**	75.85
	3e-3			53.68	53.68
	5e-5			59.79	**86.64**

Table 12: Evaluation results for two different architectures compared with BLURB leaderboard score. We compare the performance of three language models with different learning rates. Bold text shows the first and second highest score between two approaches.

PubMedQA 10 Fold			Classification Accuracy		
batch size	lr.	Language Model	BLURB	Tradition	Adapter
8	1e-5	PubMedBERT (uncased)	55.84	**67.20**	55.20
	3e-3			55.20	**62.58**
	5e-5			55.28	**55.32**
8	1e-5	BioBERT (cased)	60.24	**53.52**	55.20
	3e-3			55.20	**58.06**
	5e-5			**56.74**	55.22
8	1e-5	BERT (uncased)	51.62	53.76	**55.20**
	3e-3			55.20	55.20
	5e-5			54.02	**55.16**

Table 13: Evaluation results for two different architectures compared with BLURB leaderboard score. We compare the performance of three language models with different learning rates. Bold text shows the first and second highest score between two approaches.

can achieve the comprehensive performance with a traditional fine-tuned downstream model. However, as the BERT adapter claims that they are faster to train than the traditional approach, we found that it might not always be practical. The BERT adapter fine-tunes the downstream parameters within the language model. This causes a higher GPU usage is needed than the traditional fine-tuned downstream model, even though the weight of the language model in adapter architecture is also frozen. In the traditional fine-tuned downstream model approach, we could break down the process. We first generate data representation using a language model. This is a static representation that encodes context-dependent semantics. We can reuse this representation to fine-tune to a downstream model on top with less computation power. We believe that in some scenarios, the traditional fine-tuned downstream model approach is easier to be adopted on diverse tasks.

PubMedQA

A similar approach is repeated on the PubMedQA dataset. We compare two different architectures on different types of language models and compare the results using the accuracy metric. Results are shown in Table 13. Our

findings complement the HoC results in that the adapter approach yields equivalent performance to the traditional fine-tuned downstream model approach. However, in this setting, we notice that the adapter approach can generate better prediction results even with a high learning rate. However, as the overall accuracy of the PubMedQA is generally low, it is hard to identify the cause of the behavior. Since the adapter architecture can sometimes yield better prediction accuracy, we conclude that adapter layer parameters might be more effective, as their paper claims. This causes the downstream fine-tuning to be more compelling and worth performing more experimentation as a research of its own. We leave further exploration of this to future work.

5. Conclusion

We explored various representation learning strategies and evaluated performance of different finetuning techniques on biomedical domain tasks with different pretrained BERTmodels. We recognize that biomedical tasks can be sensitive and might need to be evaluated from various aspects rather than just an accuracy or F1 score on the leaderboard. As such research that focuses on improving performance scores might need to be aware of this delicate matter. Some of our findings have pointed out some basic but necessary limitations of BERT models that should be addressed. Being more task-oriented when handling the biomedical domain might be crucial to adopt the model on to practical tasks. We hope that further research will be more mindful of these findings.

References

Alammar, J. (n.d.-a). *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning)*. Retrieved April 26, 2021, from http://jalammar.github.io/illustrated-bert/

Alammar, J. (n.d.-b). *The Illustrated Transformer*. Retrieved April 13, 2021, from http://jalammar.github.io/illustrated-transformer/

Alsentzer, E., Murphy, J. R., Boag, W., Weng, W.-H., Jin, D., Naumann, T., & McDermott, M. B. A. (2019). Publicly Available Clinical BERT Embeddings. *ArXiv:1904.03323 [Cs]*. http://arxiv.org/abs/1904.03323

Bert-base-uncased · Hugging Face. (n.d.). Retrieved October 11, 2021, from https://huggingface.co/bert-base-uncased

BLURB Leaderboard. (n.d.). Retrieved October 11, 2021, from https://microsoft.github.io/BLURB/index.html

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171–4186. https://doi.org/10.18653/v1/N19-1423

Dmis-lab/biobert-base-cased-v1.1 · Hugging Face. (n.d.). Retrieved October 11, 2021, from https://huggingface.co/dmis-lab/biobert-base-cased-v1.1

Evolution of Natural Language Processing. (2020, September 18). Fresh Gravity. http://www.freshgravity.com/evolution-of-natural-language-processing/

Gu, Y., Tinn, R., Cheng, H., Lucas, M., Usuyama, N., Liu, X., Naumann, T., Gao, J., & Poon, H. (2021). Domain-Specific Language Model Pretraining for Biomedical Natural Language Processing. *ArXiv:2007.15779 [Cs]*. http://arxiv.org/abs/2007.15779

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., Laroussilhe, Q. D., Gesmundo, A., Attariyan, M., & Gelly, S. (2019). Parameter-Efficient Transfer Learning for NLP. *Proceedings of the 36th International Conference on Machine Learning*, 2790–2799. https://proceedings.mlr.press/v97/houlsby19a.html

Hugging Face – The AI community building the future. (n.d.). Retrieved October 11, 2021, from https://huggingface.co/

Jin, Q., Dhingra, B., Liu, Z., Cohen, W. W., & Lu, X. (2019). PubMedQA: A Dataset for Biomedical Research Question Answering. *ArXiv:1909.06146 [Cs, q-Bio]*. http://arxiv.org/abs/1909.06146

Joulin, A., Grave, E., Bojanowski, P., Douze, M., Jégou, H., & Mikolov, T. (2016). FastText.zip: Compressing text classification models. *ArXiv:1612.03651 [Cs]*. http://arxiv.org/abs/1612.03651

Lee, J., Yoon, W., Kim, S., Kim, D., Kim, S., So, C. H., & Kang, J. (2019). BioBERT: A pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, btz682. https://doi.org/10.1093/bioinformatics/btz682

Microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract · Hugging Face. (n.d.). Retrieved March 22, 2021, from https://huggingface.co/microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *ArXiv:1301.3781 [Cs]*. http://arxiv.org/abs/1301.3781

Peng, Y., Yan, S., & Lu, Z. (2019). Transfer Learning in Biomedical Natural Language Processing: An Evaluation of BERT and ELMo on Ten Benchmarking Datasets. *ArXiv:1906.05474 [Cs]*. http://arxiv.org/abs/1906.05474

Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543. https://doi.org/10.3115/v1/D14-1162

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep Contextualized Word Representations. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language*

Technologies, Volume 1 (Long Papers), 2227–2237. https://doi.org/10.18653/v1/N18-1202

Pfeiffer, J., Rücklé, A., Poth, C., Kamath, A., Vulić, I., Ruder, S., Cho, K., & Gurevych, I. (2020). AdapterHub: A Framework for Adapting Transformers. *ArXiv:2007.07779 [Cs]*. http://arxiv.org/abs/2007.07779

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. *ArXiv:1706.03762 [Cs]*. http://arxiv.org/abs/1706.03762

Vig, J. (2021). *BertViz* [Python]. https://github.com/jessevig/bertviz (Original work published 2018)

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. (2018). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, 353–355. https://doi.org/10.18653/v1/W18-5446

Yang, Z., Yang, D., Dyer, C., He, X., Smola, A., & Hovy, E. (2016). Hierarchical Attention Networks for Document Classification. *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 1480–1489. https://doi.org/10.18653/v1/N16-1174

Part 3: Appendices

Codes

1. 1_dataset_bert_transformer.py

```
# -*- coding: utf-8 -*-
"""1. Dataset BERT Transformer.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1TBUMev0Ap4h3fD53vJILQhUDoJDuCf3R

Initialization Google Drive Configuration
"""

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

"""# 1. Dataset BERT Transformer

**Created By:**  Jirarote Jirasirikul

**Monash University (Melbourne) Australia**

This file contain a code to transform input of NLP tasks (HoC and PubMedQA) from BLURB Leaderboard into BERT vector representation.
https://microsoft.github.io/BLURB/leaderboard.html

This code has been modified from www.HuggingFace.co

## Import Library

All Library and File Path will be added here
"""

# On M3 : for shell script file
# import fire

# Standard Library
import pandas as pd
import numpy as np
import glob
import os
from pathlib import Path

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, matthews_corrcoef
from sklearn.metrics import precision_recall_fscore_support,accuracy_score

from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
from collections import defaultdict

import matplotlib.pyplot as plt

from datetime import datetime
import json
# pd.set_option('display.max_colwidth', -1)

# BERT Transformer Library
!pip install transformers
import transformers as ppb

"""## Check Available Device (CPU/GPU)"""

import torch
# If there's a GPU available...
if torch.cuda.is_available():
    # Tell PyTorch to use the GPU.
    DEVICE_AVAILABLE = torch.device("cuda")
```

```
    print('There are %d GPU(s) available.' % torch.cuda.device_count())

    print('We will use the GPU:', torch.cuda.get_device_name(0))

# If not...
else:
    print('No GPU available, using the CPU instead.')
    DEVICE_AVAILABLE = torch.device("cpu")

"""## Utilities Functions"""

# Customize LOGGGING function
ENABLE_LOGS = 1
def print_log(*arg, log_type="Info"):
    global ENABLE_LOGS
    if(ENABLE_LOGS==1 or log_type!="Info"):
        print("["+log_type+"]"," ".join(str(x) for x in arg))

"""---

##BERT Text Representation

Transform Language Model

When using BERT, technically we are transforming our sentence into a vector that represent each sentence. The process is call Language Model a representation of each word.

BERT add [CLS] token infront of each sentence. This token representation vector could later be use for Classification as it contain the sentence representation.

## Define Classes

### Class : My BERT
This class build for assisting and store BERT data
"""

# BERT weight Options
# - 'distilbert-base-uncased'
# - 'bert-base-uncased'
# - 'dmis-lab/biobert-base-cased-v1.1'
# - 'dmis-lab/biobert-v1.1' : Data Mining and Information Systems Lab, Korea University's picture Updated May 19 • 41k
# - 'microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext'

class my_BERT:
    ###### Load pretrain BERT Language Model transformer (Otherwise use 'set' to customize)
    model_class, tokenizer_class, pretrained_weights = (ppb.BertModel, ppb.BertTokenizer, 'bert-base-uncased')

    # Load pretrained model/tokenizer
    bert_tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
    bert_tokenizer.add_special_tokens = True
    bert_model = model_class.from_pretrained(pretrained_weights)

    PRETRAIN_MAPPING = {'distilbert-base-uncased':'distilbert-base-uncased',
                        'bert-base-uncased':'bert-base-uncased',
                        'biobert-base-cased':'dmis-lab/biobert-base-cased-v1.1',
                        'biobert-base-uncased':'dmis-lab/biobert-v1.1',
                        'pubmedbert-base-uncased':'microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext'}

    def __init__(self, df_input,is_transform=False, ENABLE_LOGS = 1):
        ## INPUT STRUCTURE (COLUMNS):
        ## - 'text' - Required
        ## - 'label' - Optional default name is 'label' otherwise need to specific when called

        if(is_transform):
            self.df = None
            self.df_BERT = df_input
        else:
            self.df = df_input
            self.df_BERT = None
        self.ENABLE_LOGS = ENABLE_LOGS

    def print_log(self, *arg, log_type="Info"):
        if(self.ENABLE_LOGS==1 or log_type!="Info"):
```

```
        print("["+log_type+"]"," ".join(str(x) for x in arg))

    def bert_tokenize(self, token_length=128):
        df_output = self.df.copy()

        df_output['BERTTokens'] = df_output["text"].apply((lambda x: self.bert_tokenizer.encode(x,
add_special_tokens=True,truncation=True)))
        # df_output['n_tokens0'] = df_output['BERTTokens'].apply(lambda x: len(x)) # Just for verification
        temp = df_output['BERTTokens'].apply(lambda x: len(x))
        self.print_log("NO TRUNCATE","Token - Done","( mean/max no. of token:",round(temp.mean()),temp.max(),")")

        # BERT Tokenizer + truncate to BERT_MAX_LENGTH
        df_output['BERTTokens'] = df_output["text"].apply((lambda x: self.bert_tokenizer.encode(x,
add_special_tokens=True,truncation=True,max_length = token_length)))
        # df_output['n_tokens0'] = df_output['BERTTokens'].apply(lambda x: len(x)) # Just for verification
        temp = df_output['BERTTokens'].apply(lambda x: len(x))
        self.print_log("Token - Done","( mean/max no. of token:",round(temp.mean()),temp.max(),")")

        # Padding tokens to BERT_MAX_LENGTH
        df_output['BERTTokens'] = df_output['BERTTokens'].apply(lambda x: x + [0]*(token_length-len(x)))
        # df_output['n_tokens'] = df_output['BERTTokens'].apply(lambda x: len(x)) # Just for verification
        self.print_log("Pad - Done")

        # BERT Mask
        df_output['BERTMasks'] = df_output['BERTTokens'].apply(lambda x: [np.where(i != 0, 1, 0) for i in x])
        # df_output['n_mask1'] = df_output['BERTMask'].apply(lambda x: sum(x)) # Just for verification
        self.print_log("Mask - Done")

        return df_output

    def run_bert_transform(self, dataloader, device_available = torch.device("cpu")):
        all_result = []

        self.bert_model.to(device_available)

        digit = len(str(len(dataloader)))-1 # Report progress

        for step, batch in enumerate(dataloader):
            if(step == 0 or (step+1)%(10**digit) == 0 or step == len(dataloader)-1): self.print_log("Step:",step+1,"/",len(dataloader))

            b_input_ids = batch[0].to(device_available)
            b_input_mask = batch[1].to(device_available)

            with torch.no_grad():
                last_hidden_states = self.bert_model(b_input_ids, attention_mask=b_input_mask)

            res_features = last_hidden_states[0][:,0,:].cpu().numpy()
            all_result.append(res_features)
        self.print_log("BERT transform - Done")

        return np.vstack(all_result)

    def bert_transform(self, device_available = torch.device("cpu"), batch_size = 32, token_length=128):
        df_output = self.bert_tokenize(token_length)

        # Convert to Tensor
        input_tokens = torch.tensor(np.stack(df_output['BERTTokens'].values))
        input_masks = torch.tensor(np.stack(df_output['BERTMasks'].values))
        # print(input_tokens,input_masks)

        # Create the DataLoader for our training set.
        input_data = TensorDataset(input_tokens, input_masks)
        input_sampler = SequentialSampler(input_data)
        input_dataloader = DataLoader(input_data, sampler=input_sampler, batch_size=batch_size)

        self.print_log("Running BERT Transform on", str(device_available))
        if(str(device_available) == 'cpu'):
            self.print_log("Running BERT on CPU can take longer time...",log_type="WARNING")
        self.print_log("BERT token length:",token_length)
        self.print_log("Data size:",str(len(input_tokens)), "( Total batch", str(len(input_dataloader)),'* size',str(batch_size),")")

        output_features = self.run_bert_transform(input_dataloader,device_available)
        df_output = pd.concat([df_output,pd.DataFrame(output_features.tolist()).add_prefix('feature_')],axis=1)
```

```
        self.print_log("BERT transformed", log_type="Success")
        self.df_BERT = df_output

    def get_features(self):
        if(isinstance(self.df_BERT, pd.DataFrame)):
            return np.array(self.df_BERT.filter(regex='feature_',axis=1).values)
            # return np.array([np.array(xi) for xi in self.df_BERT.BERT_Features.values])
        else:
            print_log("Please run function 'bert_transform' to generate text representation first!",log_type="Error")

    def get_labels(self, list_target = ['label']):
        return np.array(self.df_BERT[list_target].values.tolist())

    def get_current_bert_model(self):
        return self.bert_model.config._name_or_path

    def load_pretrain_bert(self, model_name='bert-base-uncased'):
        ## Want BERT instead of distilBERT? Uncomment the following line:
        self.model_class, self.tokenizer_class, self.pretrained_weights = (ppb.BertModel, ppb.BertTokenizer,
self.PRETRAIN_MAPPING[model_name])

        # Load pretrained model/tokenizer
        self.bert_tokenizer = self.tokenizer_class.from_pretrained(self.pretrained_weights)
        self.bert_model = self.model_class.from_pretrained(self.pretrained_weights)

    def get_features_df(self,additional_col=[]):
        if(isinstance(self.df_BERT, pd.DataFrame)):
            return pd.concat([self.df_BERT.filter(regex='feature_',axis=1),self.df_BERT[additional_col]], axis=1)
        else:
            print_log("Please run function 'bert_transform' to generate text representation first!",log_type="Error")

"""## Data Transforming

### Hall Of Cancer (HoC)

#### Assisting function
"""

## Sample code to load raw data

# DATAPATH = "/content/drive/MyDrive/MinorThesis/"
# DATASET = "HoC"
# TOKEN_SIZE = 128
# PRETRAIN_MODEL = 'biobert-base-uncased'

# temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"train.tsv")
#     # temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"dev.tsv")
#     # temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test.tsv")

# df_train = pd.read_csv(temppath_train, sep='\t')
#     # df_test = pd.read_csv(temppath_test, sep='\t')
#     # df_valid = pd.read_csv(temppath_valid, sep='\t')

#     # TO DO : Modify this if not HoC
# df_train.columns = ['label','text','filename_line']
#     # df_test.columns = ['label','text','filename_line']
#     # df_valid.columns = ['label','text','filename_line']

"""This function help us build a context with Previous n-sentences format."""

def text_dependent(df_input, shift_level=0):
    temp_df = df_input.copy()
    new = temp_df['filename_line'].str.split("_", n = 1, expand = True)
    # making separate first name column from new data frame
    temp_df["filename"]= new[0]
    # making separate last name column from new data frame
    temp_df["sentence"]= new[1]
    if(shift_level==1):
        df_input['text'] = temp_df.groupby('filename').text.apply(lambda x: x.shift(1).fillna('')+' '+ x).str.strip()
    elif(shift_level==2):
        df_input['text'] = temp_df.groupby('filename').text.apply(lambda x: x.shift(2).fillna('')+' '+ x.shift(1).fillna('')+' '+ x).str.strip()
    elif(shift_level==3):
        df_input['text'] = temp_df.groupby('filename').text.apply(lambda x: x.shift(3).fillna('')+' '+ x.shift(2).fillna('')+' '+
x.shift(1).fillna('')+' '+ x).str.strip()
```

```
    return df_input

##### Test function
# text_dependent(df_train)

"""This function help us transform with different parameters"""

def transform_dataset(DATAPATH,DATASET,PRETRAIN_MODEL='bert-base-uncased',TOKEN_SIZE=128, SHIFT_LEVEL=None):
    temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"train.tsv")
    temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"dev.tsv")
    temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test.tsv")

    df_train = pd.read_csv(temppath_train, sep='\t')
    df_test = pd.read_csv(temppath_test, sep='\t')
    df_valid = pd.read_csv(temppath_valid, sep='\t')

    # TO DO : Modify this if not HoC
    df_train.columns = ['label','text','filename_line']
    df_test.columns = ['label','text','filename_line']
    df_valid.columns = ['label','text','filename_line']

    if(SHIFT_LEVEL != None):
        df_train = text_dependent(df_train,SHIFT_LEVEL)
        df_test = text_dependent(df_test,SHIFT_LEVEL)
        df_valid = text_dependent(df_valid,SHIFT_LEVEL)

    bert_train = my_BERT(df_train)
    bert_test = my_BERT(df_test)
    bert_valid = my_BERT(df_valid)

    bert_train.load_pretrain_bert(PRETRAIN_MODEL)
    bert_test.load_pretrain_bert(PRETRAIN_MODEL)
    bert_valid.load_pretrain_bert(PRETRAIN_MODEL)

    print_log("BERTTransform: Train Data")
    bert_train.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)
    print_log("BERTTransform: Test Data")
    bert_test.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)
    print_log("BERTTransform: Valid Data")
    bert_valid.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)

    if(SHIFT_LEVEL == None or SHIFT_LEVEL == 0):
        temp_path =
os.path.join(DATAPATH,"datasets","transformed",DATASET,PRETRAIN_MODEL,"token_length_"+str(TOKEN_SIZE))
    else:
        temp_path =
os.path.join(DATAPATH,"datasets","transformed",DATASET,PRETRAIN_MODEL,"token_length_"+str(TOKEN_SIZE)+"_shift_"+str(
SHIFT_LEVEL))
    Path(os.path.join(temp_path)).mkdir(parents=True, exist_ok=True)

    temp_df = bert_train.get_features_df(['filename_line','label'])
    temp_df.to_csv(os.path.join(temp_path,"train.csv"))
    print_log(len(df_train),"/",len(temp_df))
    temp_df = bert_test.get_features_df(['filename_line','label'])
    temp_df.to_csv(os.path.join(temp_path,"test.csv"))
    print_log(len(df_test),"/",len(temp_df))
    temp_df = bert_valid.get_features_df(['filename_line','label'])
    temp_df.to_csv(os.path.join(temp_path,"valid.csv"))
    print_log(len(df_valid),"/",len(temp_df))

    return bert_train, bert_test, bert_valid

# res = transform_dataset(DATAPATH = "/content/drive/MyDrive/MinorThesis/",
#     DATASET = "HoC",
#     TOKEN_SIZE = 512,
#     PRETRAIN_MODEL = 'biobert-base-uncased')

"""#### Executing"""

# # For Execute in M3
# if __name__ == "__main__":
#     fire.Fire(transform_dataset)

for bert_type in ['bert-base-uncased','pubmedbert-base-uncased','biobert-base-cased']:
```

```
    for shift in range(4):
        res = transform_dataset(DATAPATH = "/content/drive/MyDrive/MinorThesis/",
                                DATASET = "HoC",
                                TOKEN_SIZE = 512,
                                PRETRAIN_MODEL = bert_type,
                                SHIFT_LEVEL = shift)

## Select out of the results to Preview label count for each cancer types
# dev = res[2]
# for i in range(10):
#     print(dev.df['label'].str.split(',',expand = True)[i].value_counts())

"""### PubMedQA"""

def pubmedqa_transform_dataset(DATAPATH,DATASET,PRETRAIN_MODEL='bert-base-uncased',TOKEN_SIZE=128,
REASONING=False):
    list_data_fold = []
    for i in range(1): # We merge dataset to generate trained (Separate later using index - filename_line)
        print("fold",i)
        temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"train_set.json")
        temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"dev_set.json")

        df_temp_train = pd.read_json(temppath_train).transpose()
        df_temp_valid = pd.read_json(temppath_valid).transpose()
        list_data_fold.append((df_temp_train,df_temp_valid))
        print(df_temp_train.shape,df_temp_valid.shape)
        # print(df_temp_train.value_counts("final_decision")/450*100)
        # print(df_temp_valid.value_counts("final_decision")/50*100)

    # Test
    temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test_set.json")
    df_train = pd.concat([list_data_fold[0][0],list_data_fold[0][1]])
    df_test = pd.read_json(temppath_test).transpose()

    print("Final",df_train.shape,df_test.shape)
    return list_data_fold,df_test


# list_data_fold,df_test = pubmedqa_transform_dataset(DATAPATH = "/content/drive/MyDrive/MinorThesis/",
#             DATASET = "pubmedqa")

def pubmedqa_transform_dataset(DATAPATH,DATASET,PRETRAIN_MODEL='bert-base-uncased',TOKEN_SIZE=128,
REASONING=False,POSITION="QuesAbs"):
    # Train & Valid
    list_data_fold = []
    for i in range(1): # We merge dataset to generate trained (Separate later using index - filename_line)
        temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"train_set.json")
        temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"dev_set.json")

        df_temp_train = pd.read_json(temppath_train).transpose()
        df_temp_valid = pd.read_json(temppath_valid).transpose()
        list_data_fold.append((df_temp_train,df_temp_valid))
        print(df_temp_train.shape,df_temp_valid.shape)

    # Test
    temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test_set.json")
    df_train = pd.concat([list_data_fold[0][0],list_data_fold[0][1]])
    df_test = pd.read_json(temppath_test).transpose()

    print("Final",df_train.shape,df_test.shape)

    # # TO DO : Modify this if not HoC


    if(REASONING):
        # REASONING REQUIRED
        df_train_mod =
df_train[['QUESTION','CONTEXTS','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
        if(POSITION == "QuesAbs"):
            df_train_mod['text'] = df_train_mod.QUESTION +". "+ df_train_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question
before
        else:
            df_train_mod['text'] = df_train_mod.QUESTION +". "+ df_train_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question
before
```

```
        # df_train_mod['text'] = df_train_mod.CONTEXTS.apply(lambda x : (' ').join(x)) +" "+ df_train_mod.QUESTION  #question after
        df_train_mod.drop(columns=['QUESTION','CONTEXTS'],inplace=True)
        df_train_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']

        df_test_mod =
df_test[['QUESTION','CONTEXTS','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
        if(POSITION == "QuesAbs"):
            df_test_mod['text'] = df_test_mod.QUESTION +". "+ df_test_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question before
        else:
            df_test_mod['text'] = df_test_mod.CONTEXTS.apply(lambda x : (' ').join(x)) +" "+ df_test_mod.QUESTION  #question after

        df_test_mod.drop(columns=['QUESTION','CONTEXTS'],inplace=True)
        df_test_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']
    else:
        # REASONING FREE
        df_train_mod =
df_train[['QUESTION','LONG_ANSWER','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
        if(POSITION == "QuesAbs"):
            df_train_mod['text'] = df_train_mod.QUESTION +" "+ df_train_mod.LONG_ANSWER #question before
        else:
            df_train_mod['text'] = df_train_mod.LONG_ANSWER +" "+ df_train_mod.QUESTION #question after

        df_train_mod.drop(columns=['QUESTION','LONG_ANSWER'],inplace=True)
        df_train_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']

        df_test_mod =
df_test[['QUESTION','LONG_ANSWER','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
        if(POSITION == "QuesAbs"):
            df_test_mod['text'] = df_test_mod.QUESTION +" "+ df_test_mod.LONG_ANSWER #question before
        else:
            df_test_mod['text'] = df_test_mod.LONG_ANSWER +" "+ df_test_mod.QUESTION #question after

        df_test_mod.drop(columns=['QUESTION','LONG_ANSWER'],inplace=True)
        df_test_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']


    bert_train = my_BERT(df_train_mod)
    bert_test = my_BERT(df_test_mod)

    bert_train.load_pretrain_bert(PRETRAIN_MODEL)
    bert_test.load_pretrain_bert(PRETRAIN_MODEL)

    print_log("BERTTransform: Train Data")
    bert_train.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)
    print_log("BERTTransform: Test Data")
    bert_test.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)

    temp_path = os.path.join(DATAPATH,"datasets","transformed",DATASET,"QuesAbs","reasoning_required" if REASONING else
"reasoning_free",PRETRAIN_MODEL,"token_length_"+str(TOKEN_SIZE))
    Path(os.path.join(temp_path)).mkdir(parents=True, exist_ok=True)

    temp_df = bert_train.get_features_df(['id','label','reasoning_required_pred','reasoning_free_pred'])
    temp_df.to_csv(os.path.join(temp_path,"train.csv"))
    print_log(len(df_train),"/",len(temp_df))
    temp_df = bert_test.get_features_df(['id','label','reasoning_required_pred','reasoning_free_pred'])
    temp_df.to_csv(os.path.join(temp_path,"test.csv"))
    print_log(len(df_test),"/",len(temp_df))

    return bert_train, bert_test

# df_train,df_test = pubmedqa_transform_dataset(DATAPATH = "/content/drive/MyDrive/MinorThesis/",
#             DATASET = "pubmedqa",
#             TOKEN_SIZE = 512,
#             PRETRAIN_MODEL = 'pubmedbert-base-uncased',
#             REASONING=True)

for i in [True,False]:
    for j in ['pubmedbert-base-uncased','bert-base-uncased','biobert-base-cased']:
        df_train,df_test = pubmedqa_transform_dataset(DATAPATH = "/content/drive/MyDrive/MinorThesis/",
            DATASET = "pubmedqa",
            TOKEN_SIZE = 512,
            PRETRAIN_MODEL = j,
            REASONING=i)
```

```
# # For test in Colab Only
# datapath = "/content/drive/MyDrive/MinorThesis/"
# dataset = "pubmedqa"
# token_size = 128
# pretrain_model = 'biobert-base-uncased'

# # Train & Valid
# list_data_fold = []
# for i in range(10):
#     temppath_train = os.path.join(datapath,"datasets","raw",dataset,"pqal_fold"+str(i),"train_set.json")
#     temppath_valid = os.path.join(datapath,"datasets","raw",dataset,"pqal_fold"+str(i),"dev_set.json")

#     df_temp_train = pd.read_json(temppath_train).transpose()
#     df_temp_valid = pd.read_json(temppath_valid).transpose()
#     list_data_fold.append((df_temp_train,df_temp_valid))
#     print(df_temp_train.shape,df_temp_valid.shape)

# # Test
# temppath_test = os.path.join(datapath,"datasets","raw",dataset,"test_set.json")
# df_test = pd.read_json(temppath_test).transpose()

# # Test Label
# temppath_test_gt = os.path.join(datapath,"datasets","raw",dataset,"test_ground_truth.json")
# df_test_label = pd.read_json(temppath_test_gt, typ='series')

# # # Original 1k data before split
# # temppath_ori = os.path.join(datapath,"datasets","raw",dataset,"ori_pqal.json")
# # df_ori = pd.read_json(temppath_ori).transpose()
# # df_ori.shape

# print("Final",list_data_fold[0][0].shape,list_data_fold[0][1].shape,df_test.shape)

# # Extract Fold ID (test,train)

# def get_pubmedqa_fold_id():
#     list_data_fold = []
#     for i in range(10):
#         temppath_train = os.path.join(datapath,"datasets","raw",dataset,"pqal_fold"+str(i),"train_set.json")
#         temppath_valid = os.path.join(datapath,"datasets","raw",dataset,"pqal_fold"+str(i),"dev_set.json")

#         df_temp_train = pd.read_json(temppath_train).transpose().reset_index()
#         df_temp_valid = pd.read_json(temppath_valid).transpose().reset_index()

#         list_data_fold.append((df_temp_train['index'].values,df_temp_valid['index'].values))
#         # list_data_fold.append((df_temp_train,df_temp_valid))
#         # print(df_temp_train.shape,df_temp_valid.shape)
#     return list_data_fold

# list_data_fold = get_pubmedqa_fold_id()

"""### BioASQ"""

def bioasq_transform_dataset(DATAPATH,DATASET,PRETRAIN_MODEL='bert-base-uncased',TOKEN_SIZE=128,
REASONING=False):
    # Train & Valid & Test
    temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"train.tsv")
    temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"dev.tsv")
    temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test.tsv")

    df_train = pd.read_csv(temppath_train,sep="\t",header=None)
    df_valid = pd.read_csv(temppath_valid,sep="\t",header=None)
    df_test = pd.read_csv(temppath_test,sep="\t",header=None)

    df_train.columns = ["id","question","answer","label"]
    df_valid.columns = ["id","question","answer","label"]
    df_test.columns = ["id","question","answer","label"]

    print("Final",df_train.shape,df_valid.shape,df_test.shape)

    # # TO DO : Modify this if not HoC
    df_train['text'] = df_train.question +". "+ df_train.answer
    df_valid['text'] = df_valid.question +". "+ df_valid.answer
    df_test['text'] = df_test.question +". "+ df_test.answer
```

```
    bert_train = my_BERT(df_train)
    bert_test = my_BERT(df_test)
    bert_valid = my_BERT(df_valid)

    bert_train.load_pretrain_bert(PRETRAIN_MODEL)
    bert_test.load_pretrain_bert(PRETRAIN_MODEL)
    bert_valid.load_pretrain_bert(PRETRAIN_MODEL)

    print_log("BERTTransform: Train Data")
    bert_train.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)
    print_log("BERTTransform: Test Data")
    bert_test.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)
    print_log("BERTTransform: Valid Data")
    bert_valid.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)

    temp_path =
os.path.join(DATAPATH,"datasets","transformed",DATASET,PRETRAIN_MODEL,"token_length_"+str(TOKEN_SIZE))
    Path(os.path.join(temp_path)).mkdir(parents=True, exist_ok=True)

    # Additional labels for HOC

    temp_df = bert_train.get_features_df(['id','label'])
    temp_df.to_csv(os.path.join(temp_path,"train.csv"))
    print_log(len(df_train),"/",len(temp_df))
    temp_df = bert_test.get_features_df(['id','label'])
    temp_df.to_csv(os.path.join(temp_path,"test.csv"))
    print_log(len(df_test),"/",len(temp_df))
    temp_df = bert_valid.get_features_df(['id','label'])
    temp_df.to_csv(os.path.join(temp_path,"valid.csv"))
    print_log(len(df_valid),"/",len(temp_df))

    # return bert_train, bert_test

# bioasq_transform_dataset(DATAPATH = "/content/drive/MyDrive/MinorThesis/",
#             DATASET = "BioASQ",
#             TOKEN_SIZE = 512,
#             PRETRAIN_MODEL = 'pubmedbert-base-uncased',
#             REASONING=True)

# # For test in Colab Only
# datapath = "/content/drive/MyDrive/MinorThesis/"
# dataset = "BioASQ"
# token_size = 512
# pretrain_model = 'biobert-base-uncased'

# # Train & Valid
# temppath_train = os.path.join(datapath,"datasets","raw",dataset,"train.tsv")
# temppath_valid = os.path.join(datapath,"datasets","raw",dataset,"dev.tsv")
# temppath_test = os.path.join(datapath,"datasets","raw",dataset,"test.tsv")

# df_train = pd.read_csv(temppath_train,sep="\t",header=None)
# df_valid = pd.read_csv(temppath_valid,sep="\t",header=None)
# df_test = pd.read_csv(temppath_test,sep="\t",header=None)

# df_train.columns = ["id","question","answer","label"]
# df_valid.columns = ["id","question","answer","label"]
# df_test.columns = ["id","question","answer","label"]

# df_train
```

2. 2_downstream_model_generator.py

```
# -*- coding: utf-8 -*-
"""2. Downstream Model Generator.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1OGPWNsUMrdIzTTEUf1ZyJXad_P_BoA0M

Initialization Google Drive Configuration
"""

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

"""# 2. Downstream Model Generator

**Created By:**  Jirarote Jirasirikul

**Monash University (Melbourne) Australia**

## Import Library

All Library and File Path will be added here
"""

# Standard Library
import pandas as pd
import numpy as np
import glob
import os
from pathlib import Path

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, matthews_corrcoef
from sklearn.metrics import precision_recall_fscore_support,accuracy_score, classification_report
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
from collections import defaultdict

import matplotlib.pyplot as plt

from datetime import datetime
import json

import seaborn as sns
from tqdm.notebook import tqdm

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler

# BERT Transformer Library
!pip install transformers
import transformers as ppb

"""## Check Available Device (CPU/GPU)"""

import torch
# If there's a GPU available...
if torch.cuda.is_available():
    # Tell PyTorch to use the GPU.
    DEVICE_AVAILABLE = torch.device("cuda")

    print('There are %d GPU(s) available.' % torch.cuda.device_count())

    print('We will use the GPU:', torch.cuda.get_device_name(0))
```

```
# If not...
else:
    print('No GPU available, using the CPU instead.')
    DEVICE_AVAILABLE = torch.device("cpu")

"""## Utilities Functions"""

# # MY GLOBAL FUNCTION -

ENABLE_LOGS = 1
def print_log(*arg, log_type="Info"):
    global ENABLE_LOGS
    if(ENABLE_LOGS==1 or log_type!="Info"):
        print("["+log_type+"]"," ".join(str(x) for x in arg))

"""---

##BERT Text Representation

Transform Language Model

When using BERT, technically we are transforming our sentence into a vector that represent each sentence. The process is call Language Model a representation of each word.

BERT add [CLS] token infront of each sentence. This token representation vector could later be use for Classification as it contain the sentence representation.

### Class my_BERT
"""

# BERT weight Options
# - 'distilbert-base-uncased'
# - 'bert-base-uncased'
# - 'dmis-lab/biobert-base-cased-v1.1'
# - 'dmis-lab/biobert-v1.1' : Data Mining and Information Systems Lab, Korea University's picture Updated May 19 • 41k
# - 'microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext'

class my_BERT:
    ###### Load pretrain BERT Language Model transformer (Otherwise use 'set' to customize)
    model_class, tokenizer_class, pretrained_weights = (ppb.BertModel, ppb.BertTokenizer, 'bert-base-uncased')

    # Load pretrained model/tokenizer
    bert_tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
    bert_tokenizer.add_special_tokens = True
    bert_model = model_class.from_pretrained(pretrained_weights)

    PRETRAIN_MAPPING = {'distilbert-base-uncased':'distilbert-base-uncased',
                        'bert-base-uncased':'bert-base-uncased',
                        'biobert-base-cased':'dmis-lab/biobert-base-cased-v1.1',
                        'biobert-base-uncased':'dmis-lab/biobert-v1.1',
                        'pubmedbert-base-uncased':'microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext'}

    def __init__(self, df_input,is_transform=False, ENABLE_LOGS = 1):
        ## INPUT STRUCTURE (COLUMNS):
        ## - 'text' - Required
        ## - 'label' - Optional default name is 'label' otherwise need to specific when called

        if(is_transform):
            self.df = None
            self.df_BERT = df_input
        else:
            self.df = df_input
            self.df_BERT = None
        self.ENABLE_LOGS = ENABLE_LOGS

    def print_log(self, *arg, log_type="Info"):
        if(self.ENABLE_LOGS==1 or log_type!="Info"):
            print("["+log_type+"]"," ".join(str(x) for x in arg))

    def bert_tokenize(self, token_length=128):
        df_output = self.df.copy()

        # BERT Tokenizer + truncate to BERT_MAX_LENGTH
```

```
        df_output['BERTTokens'] = df_output["text"].apply((lambda x: self.bert_tokenizer.encode(x,
add_special_tokens=True,truncation=True,max_length = token_length)))
        # df_output['n_tokens0'] = df_output['BERTTokens'].apply(lambda x: len(x)) # Just for verification
        temp = df_output['BERTTokens'].apply(lambda x: len(x))
        self.print_log("Token - Done","( mean/max no. of token:",round(temp.mean()),temp.max(),")")

        # Padding tokens to BERT_MAX_LENGTH
        df_output['BERTTokens'] = df_output['BERTTokens'].apply(lambda x: x + [0]*(token_length-len(x)))
        # df_output['n_tokens'] = df_output['BERTTokens'].apply(lambda x: len(x)) # Just for verification
        self.print_log("Pad - Done")

        # BERT Mask
        df_output['BERTMasks'] = df_output['BERTTokens'].apply(lambda x: [np.where(i != 0, 1, 0) for i in x])
        # df_output['n_mask1'] = df_output['BERTMask'].apply(lambda x: sum(x)) # Just for verification
        self.print_log("Mask - Done")

        return df_output

    def run_bert_transform(self, dataloader, device_available = torch.device("cpu")):
        all_result = []

        self.bert_model.to(device_available)

        digit = len(str(len(dataloader)))-1 # Report progress

        for step, batch in enumerate(dataloader):
            if(step == 0 or (step+1)%(10**digit) == 0 or step == len(dataloader)-1): self.print_log("Step:",step+1,"/",len(dataloader))

            b_input_ids = batch[0].to(device_available)
            b_input_mask = batch[1].to(device_available)

            with torch.no_grad():
                last_hidden_states = self.bert_model(b_input_ids, attention_mask=b_input_mask)


            res_features = last_hidden_states[0][:,0,:].cpu().numpy()
            all_result.append(res_features)
        self.print_log("BERT transform - Done")

        return np.vstack(all_result)


    def bert_transform(self, device_available = torch.device("cpu"), batch_size = 32, token_length=128):
        df_output = self.bert_tokenize(token_length)

        # Convert to Tensor
        input_tokens = torch.tensor(np.stack(df_output['BERTTokens'].values))
        input_masks = torch.tensor(np.stack(df_output['BERTMasks'].values))
        # print(input_tokens,input_masks)

        # Create the DataLoader for our training set.
        input_data = TensorDataset(input_tokens, input_masks)
        input_sampler = SequentialSampler(input_data)
        input_dataloader = DataLoader(input_data, sampler=input_sampler, batch_size=batch_size)

        self.print_log("Running BERT Transform on", str(device_available))
        if(str(device_available) == 'cpu'):
            self.print_log("Running BERT on CPU can take longer time...",log_type="WARNING")
        self.print_log("BERT token length:",token_length)
        self.print_log("Data size:",str(len(input_tokens)), "( Total batch", str(len(input_dataloader)),'* size',str(batch_size),")")

        output_features = self.run_bert_transform(input_dataloader,device_available)
        df_output = pd.concat([df_output,pd.DataFrame(output_features.tolist()).add_prefix('feature_')],axis=1)

        self.print_log("BERT transformed", log_type="Success")
        self.df_BERT = df_output

    def get_features(self):
        if(isinstance(self.df_BERT, pd.DataFrame)):
            return np.array(self.df_BERT.filter(regex='feature_',axis=1).values)
            # return np.array([np.array(xi) for xi in self.df_BERT.BERT_Features.values])
        else:
            print_log("Please run function 'bert_transform' to generate text representation first!",log_type="Error")
```

```
    def get_labels(self, list_target = ['label']):
        return np.array(self.df_BERT[list_target].values.tolist())

    def get_current_bert_model(self):
        return self.bert_model.config._name_or_path

    def load_pretrain_bert(self, model_name='bert-base-uncased'):
        ## Want BERT instead of distilBERT? Uncomment the following line:
        self.model_class, self.tokenizer_class, self.pretrained_weights = (ppb.BertModel, ppb.BertTokenizer,
self.PRETRAIN_MAPPING[model_name])

        # Load pretrained model/tokenizer
        self.bert_tokenizer = self.tokenizer_class.from_pretrained(self.pretrained_weights)
        self.bert_model = self.model_class.from_pretrained(self.pretrained_weights)

    def get_features_df(self,additional_col=[]):
        if(isinstance(self.df_BERT, pd.DataFrame)):
            return pd.concat([self.df_BERT.filter(regex='feature_',axis=1),self.df_BERT[additional_col]], axis=1)
        else:
            print_log("Please run function 'bert_transform' to generate text representation first!",log_type="Error")

    ### BELOW is an EXTENSION

    def extract_hoc_label(self):
        try:
            LABEL_LIST = ['label_IM', 'label_ID', 'label_CE', 'label_RI', 'label_GS', 'label_GI', 'label_A', 'label_CD', 'label_PS', 'label_TPI']
            temp = self.df_BERT['label'].str.split(',').apply(lambda x: [int(i.split('_')[1]) for i in x])
            temp_df = pd.DataFrame(temp.tolist())
            temp_df.columns = LABEL_LIST
            self.df_BERT = pd.concat([self.df_BERT,temp_df], axis=1)
            print_log("Extract HoC label",log_type="Success")
            return self.df_BERT
        except:
            print_log("Something went wrong",log_type="Error")

    def class2idx_pubmedqa_label(self,col='label'):
        try:
            class2idx = {
                'no':0,
                'maybe':1,
                'yes':2
            }
            # idx2class = {v: k for k, v in class2idx.items()}
            self.df_BERT[col].replace(class2idx, inplace=True)
            print_log("class2idx_pubmedqa_label",log_type="Success")
            return self.df_BERT
        except:
            print_log("Something went wrong",log_type="Error")

    def class2idx_bioasq_label(self,col='label'):
        try:
            class2idx = {
                'no':0,
                'yes':1
            }
            # idx2class = {v: k for k, v in class2idx.items()}
            self.df_BERT[col].replace(class2idx, inplace=True)
            print_log("class2idx_bioasq_label",log_type="Success")
            return self.df_BERT
        except:
            print_log("Something went wrong",log_type="Error")

"""---


## Downstream Model

### Class: my_downstream

Dataset training hyperparameter in BLURB Paper
*   learning_rate = [1e-5, 3e-3, 5e-5]
*   batch_size = (16,32)
*   epoch_number = 2~60
"""
```

```
class LinearLayer(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(LinearLayer, self).__init__()

        self.linear = torch.nn.Linear(D_in, D_out)
        self.dropout = torch.nn.Dropout(0.1)

    def forward(self, x):
        """
        In the forward function we accept a Tensor of input data and we must return
        a Tensor of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Tensors.
        """
        output = self.linear(x)
        # output = self.dropout(output)
        output = torch.sigmoid(output)
        return output

class MulticlassClassification(torch.nn.Module):
    def __init__(self, num_feature, num_class):
        super(MulticlassClassification, self).__init__()

        self.layer_1 = nn.Linear(num_feature, 512)
        self.layer_2 = nn.Linear(512, 128)
        self.layer_3 = nn.Linear(128, 64)
        self.layer_out = nn.Linear(64, num_class)

        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=0.2)
        self.batchnorm1 = nn.BatchNorm1d(512)
        self.batchnorm2 = nn.BatchNorm1d(128)
        self.batchnorm3 = nn.BatchNorm1d(64)

    def forward(self, x):
        x = self.layer_1(x)
        x = self.batchnorm1(x)
        x = self.relu(x)

        x = self.layer_2(x)
        x = self.batchnorm2(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.layer_3(x)
        x = self.batchnorm3(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.layer_out(x)

        return x

class my_downstream:
    def __init__(self, train_x, train_y, test_x = None, test_y = None, valid_x = None, valid_y = None, ENABLE_LOGS = 1):
        self.train_x = train_x
        self.train_y = train_y
        self.test_x = test_x
        self.test_y = test_y
        self.valid_x = valid_x
        self.valid_y = valid_y
        self.predict_x = None
        self.predict_y = None

        self.model = None
        self.best_model = None
        self.best_iter = None

        self.ENABLE_LOGS = ENABLE_LOGS


        self.accuracy_stats = {
                            'train': [],
                            "val": []
```

```
                    }
    self.loss_stats = {
                    'train': [],
                    "val": []
                }

def print_log(self, *arg, log_type="Info"):
    if(self.ENABLE_LOGS==1 or log_type!="Info"):
        print("["+log_type+"]"," ".join(str(x) for x in arg))

def train_logistic(self, n_iter=500, random_state=0):
    self.model = LogisticRegression(random_state=random_state,max_iter=n_iter)
    self.print_log("Train", self.model.__class__.__name__)
    self.print_log("iteration:",n_iter)
    self.print_log("random_state:",random_state)
    self.model.fit(self.train_x, self.train_y)
    self.print_log("Train Logistic Regression", log_type="Success")

def train_multiclass_nn(self, D_in, n_classes, EPOCHS=500, learning_rate = 1e-5, batch_size = 32, CONT = 0):
    class_count = [i for i in get_class_distribution(self.train_y).values()]
    class_weights = 1./torch.tensor(class_count, dtype=torch.float)
    # print(class_weights)


    self.model = MulticlassClassification(num_feature = D_in, num_class=n_classes).to(DEVICE_AVAILABLE)

    criterion = nn.CrossEntropyLoss(weight=class_weights.to(DEVICE_AVAILABLE))
    optimizer = optim.Adam(self.model.parameters(), lr=learning_rate)
    # print(self.model)

    # Convert to Tensor
    input_x = torch.tensor(self.train_x).float()
    input_y = torch.tensor(self.train_y).long()
    valid_x = torch.tensor(self.valid_x).float()
    valid_y = torch.tensor(self.valid_y).long()

    # Create the DataLoader for our training set.
    input_data = TensorDataset(input_x, input_y)
    input_sampler = SequentialSampler(input_data)
    input_dataloader = DataLoader(input_data, sampler=input_sampler, batch_size=batch_size)
    self.print_log("Data size:",str(len(self.train_x)), "( Total batch", str(len(input_dataloader)),'* size',str(batch_size),")")
    valid_data = TensorDataset(valid_x, valid_y)
    valid_sampler = SequentialSampler(valid_data)
    valid_dataloader = DataLoader(valid_data, sampler=valid_sampler, batch_size=batch_size)
    self.print_log("Data size:",str(len(self.valid_x)), "( Total batch", str(len(valid_dataloader)),'* size',str(batch_size),")")

    min_loss = 999999

    self.print_log("Begin training.")
    for e in tqdm(range(1, EPOCHS+1)):
        # TRAINING
        train_epoch_loss = 0
        train_epoch_acc = 0

        self.model.train()
        for X_train_batch, y_train_batch in input_dataloader:
            X_train_batch, y_train_batch = X_train_batch.to(DEVICE_AVAILABLE), y_train_batch.to(DEVICE_AVAILABLE)

            optimizer.zero_grad()

            y_train_pred = self.model(X_train_batch)
            train_loss = criterion(y_train_pred, y_train_batch.squeeze_())
            train_acc = my_evaluator.multi_acc(y_train_pred, y_train_batch)

            train_loss.backward()
            optimizer.step()

            train_epoch_loss += train_loss.item()
            train_epoch_acc += train_acc.item()

        # VALIDATION
        with torch.no_grad():

            val_epoch_loss = 0
```

```
            val_epoch_acc = 0

            self.model.eval()
            for X_val_batch, y_val_batch in valid_dataloader:
                X_val_batch, y_val_batch = X_val_batch.to(DEVICE_AVAILABLE), y_val_batch.to(DEVICE_AVAILABLE)

                y_val_pred = self.model(X_val_batch)

                val_loss = criterion(y_val_pred, y_val_batch.squeeze_())
                val_acc = my_evaluator.multi_acc(y_val_pred, y_val_batch)

                val_epoch_loss += val_loss.item()
                val_epoch_acc += val_acc.item()
        self.loss_stats['train'].append(train_epoch_loss/len(input_dataloader))
        self.loss_stats['val'].append(val_epoch_loss/len(valid_dataloader))
        self.accuracy_stats['train'].append(train_epoch_acc/len(input_dataloader))
        self.accuracy_stats['val'].append(val_epoch_acc/len(valid_dataloader))

        if(min_loss > self.loss_stats['val'][-1]):
            self.print_log(f'Epoch {e+0:03}: SAVE')
            self.best_model = self.model
            self.best_iter = e+1
            min_loss = self.loss_stats['val'][-1]

        # print(f'Epoch {e+0:03}: | Train Loss: {train_epoch_loss/len(input_dataloader):.5f} | Val Loss:
{val_epoch_loss/len(valid_dataloader):.5f} | Train Acc: {train_epoch_acc/len(input_dataloader):.3f}| Val Acc:
{val_epoch_acc/len(valid_dataloader):.3f}')

    def train_linear_nn(self, D_in, n_classes, n_iter=500, learning_rate = 1e-5, batch_size = 32, CONT = 0):
        list_train_loss = []
        list_valid_loss = []

        # Convert to Tensor
        input_x = torch.tensor(self.train_x).float()
        input_y = torch.tensor(self.train_y).float()

        # Create the DataLoader for our training set.
        input_data = TensorDataset(input_x, input_y)
        input_sampler = SequentialSampler(input_data)
        input_dataloader = DataLoader(input_data, sampler=input_sampler, batch_size=batch_size)
        self.print_log("Data size:",str(len(self.train_x)), "( Total batch", str(len(input_dataloader)),'* size',str(batch_size),")")

        if(CONT == 0):
            self.model = LinearLayer(D_in, n_classes).to(DEVICE_AVAILABLE)
        self.print_log("Train", self.model.__class__.__name__)
        self.print_log("iteration:",n_iter)

        criterion = torch.nn.BCELoss()
        optimizer = torch.optim.Adam(self.model.parameters(), lr=learning_rate)

        min_loss = 999999

        digit_iter = len(str(n_iter))-1 # Report progress
        for i in range(n_iter):
            digit = len(str(len(input_dataloader)))-1 # Report progress
            for step, batch in enumerate(input_dataloader):

                x_input = batch[0].to(DEVICE_AVAILABLE)
                y_input = batch[1].to(DEVICE_AVAILABLE)
                y_pred = self.model(x_input).squeeze()

                loss = criterion(y_pred,y_input)

                # if(step == 0 or (step+1)%(10**digit) == 0 or step == len(input_dataloader)-1):
                #     self.print_log("Step:",step+1,"/",len(input_dataloader),":",loss.item())

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

            # Train
            x_train = torch.tensor(self.train_x).float().to(DEVICE_AVAILABLE)
            y_train = torch.tensor(self.train_y).float().to(DEVICE_AVAILABLE)
```

```
        with torch.no_grad():
            y_vpred_train = self.model(x_train).squeeze()

        iter_loss_train = criterion(y_vpred_train, y_train)
        list_train_loss.append(iter_loss_train.item())

        # Valid
        x_valid = torch.tensor(self.valid_x).float().to(DEVICE_AVAILABLE)
        y_valid = torch.tensor(self.valid_y).float().to(DEVICE_AVAILABLE)

        with torch.no_grad():
            y_vpred_valid = self.model(x_valid).squeeze()

        iter_loss_valid = criterion(y_vpred_valid, y_valid)
        list_valid_loss.append(iter_loss_valid.item())

        if(min_loss > iter_loss_valid.item()):
            self.print_log("Iteration:",i+1,"/",n_iter,":",'train',iter_loss_train.item(),'valid',iter_loss_valid.item())
            self.best_model = self.model
            self.best_iter = i+1
            min_loss = iter_loss_valid.item()
        # else:
        #     break

    self.print_log("Train Linear NN", log_type="Success")
    return list_train_loss, list_valid_loss

def get_model_name(self):
    if(self.model is None):
        return "None"
    else:
        return self.model.__class__.__name__

def predict(self, test_x = None):
    # Train Data
    self.predict_x = self.test_x if test_x is None else test_x
    if(self.predict_x is None): return print_log("No test data", log_type="Error")

    # Model
    if(self.model is None):
        print_log("Predict","NULL MODEL", log_type="WARNING")
        self.predict_y = [0]*len(self.predict_x)
    elif(self.get_model_name() == 'LogisticRegression'):
        print_log("Predict",self.get_model_name())
        self.predict_y = self.model.predict(self.predict_x)
    elif(self.get_model_name() == 'LinearLayer'):
        print_log("Predict",self.get_model_name())
        input = torch.tensor(self.predict_x).float().to(DEVICE_AVAILABLE)

        with torch.no_grad():
            self.predict_y = self.best_model(input).squeeze().round().cpu().detach().numpy()
    elif(self.get_model_name() == 'MulticlassClassification'):
        print_log("Predict",self.get_model_name())
        # Convert to Tensor
        input_x = torch.tensor(self.predict_x).float()
        input_y = torch.tensor(self.test_y).long()

        # Create the DataLoader for our training set.
        input_data = TensorDataset(input_x, input_y)
        input_sampler = SequentialSampler(input_data)
        input_dataloader = DataLoader(input_data, sampler=input_sampler, batch_size=1)
        self.print_log("Data size:",str(len(self.train_x)), "( Total batch", str(len(input_dataloader)),'* size',str(1),")")

        y_pred_list = []
        with torch.no_grad():
            self.best_model.eval()
            for X_batch, _ in input_dataloader:
                X_batch = X_batch.to(DEVICE_AVAILABLE)
                y_test_pred = self.best_model(X_batch)
                _, y_pred_tags = torch.max(y_test_pred, dim = 1)
                y_pred_list.append(y_pred_tags.cpu().numpy())
        self.predict_y = [a.squeeze().tolist() for a in y_pred_list]
    else:
        print_log("Predict",self.get_model_name())
```

```
            print_log("something wrong with prediction function", log_type="ERROR")

        return self.predict_x,self.predict_y

    def __str__(self):
        return self.get_model_name()

    def load_linear_nn(self, D_in, n_classes, model_path):
        self.model = LinearLayer(D_in, n_classes).to(DEVICE_AVAILABLE)
        self.model.load_state_dict(torch.load(model_path, map_location=DEVICE_AVAILABLE))
        self.best_model = self.model

    def load_multiclass_nn(self, D_in, n_classes, model_path):
        self.model = MulticlassClassification(num_feature = D_in, num_class=n_classes).to(DEVICE_AVAILABLE)
        self.model.load_state_dict(torch.load(model_path, map_location=DEVICE_AVAILABLE))
        self.best_model = self.model

# labels = ['label_IM', 'label_ID', 'label_CE', 'label_RI', 'label_GS', 'label_GI', 'label_A', 'label_CD', 'label_PS', 'label_TPI']
# learning_rate = 1e-5
# batch_size = 32

# print_log("Training Downstream Model")
# model = my_downstream(bert_train.get_features(),bert_train.get_labels(labels),
#                       bert_test.get_features(),bert_test.get_labels(labels),
#                       bert_valid.get_features(),bert_valid.get_labels(labels))

# train_loss, valid_loss = model.train_linear_nn(D_in = 768,
#                       n_classes = 1 if type(labels) == str else len(labels),
#                       n_iter=100,
#                       learning_rate = learning_rate,
#                       batch_size = batch_size)

# _,predict_y = model.predict()

"""### Class: my_evaluator"""

class my_evaluator:
    LABELS_HOC_FULL = ['activating invasion and metastasis', 'avoiding immune destruction',
                'cellular energetics', 'enabling replicative immortality', 'evading growth suppressors',
                'genomic instability and mutation', 'inducing angiogenesis', 'resisting cell death',
                'sustaining proliferative signaling', 'tumor promoting inflammation']
    LABELS_HOC_SHORT = ['label_IM', 'label_ID',
                    'label_CE', 'label_RI', 'label_GS',
                    'label_GI', 'label_A', 'label_CD',
                    'label_PS', 'label_TPI']
    @classmethod
    def divide(self, x, y):
        return np.true_divide(x, y, out=np.zeros_like(x, dtype=np.float), where=y != 0)

    @classmethod
    def get_p_r_f_arrary(self, test_predict_label, test_true_label):
        num, cat = test_predict_label.shape
        # print(num,cat)
        acc_list = []
        prc_list = []
        rec_list = []
        f_score_list = []
        for i in range(num):
            # print(test_predict_label[i])
            # print(test_true_label[i])

            acc = accuracy_score(test_true_label[i], test_predict_label[i])
            prc,rec,f_score,_ = precision_recall_fscore_support(test_true_label[i], test_predict_label[i], average='macro')

            if prc == 0 and rec == 0:
                f_score = 0
            else:
                f_score = 2 * prc * rec / (prc + rec)

            acc_list.append(acc)
            prc_list.append(prc)
            rec_list.append(rec)
            f_score_list.append(f_score)
```

```
    # print(prc_list)
    # print(rec_list)

    mean_prc = np.mean(prc_list)
    mean_rec = np.mean(rec_list)
    f_score = self.divide(2 * mean_prc * mean_rec, (mean_prc + mean_rec))
    return mean_prc, mean_rec, f_score

  @classmethod
  def hoc_sentence2doc(self, input_df):
    # Output variables
    data = {}
    input_labels_count = dict(zip(self.LABELS_HOC_FULL, [0]*len(self.LABELS_HOC_FULL))) # sentence

    # Group sentence back into documents
    for i in range(len(input_df)):
      input_row = input_df.iloc[i]

      key = input_row['index'][:input_row['index'].find('_')]

      if key not in data:
        data[key] = set()

      if not pd.isna(input_row['labels']):
        for l in input_row['labels'].split(','):
          ind,val = l.split('_')
          if(val != '0'):
            data[key].add(self.LABELS_HOC_FULL[int(ind)])
            input_labels_count[self.LABELS_HOC_FULL[int(ind)]] += 1

    return data, input_labels_count

  @classmethod
  def hoc_labels2np(self, data):
    labels_list = dict(zip(self.LABELS_HOC_FULL, [[],[],[],[],[],[],[],[],[],[]]))

    y_np = []

    for k, v in data.items():
      # print(k)
      # print(true,pred)
      t = [0] * len(self.LABELS_HOC_FULL)
      for i in v:
        t[self.LABELS_HOC_FULL.index(i)] = 1

      y_np.append(t)

      for lab in self.LABELS_HOC_FULL:
        if(lab in v):
          labels_list[lab].append(1)
        else:
          labels_list[lab].append(0)

    return np.array(y_np),labels_list

  @classmethod
  def analysis_hoc(self, input_df):
    # Reformat to Paper evaluator format
    input_df = input_df[['filename_line','label']].copy()
    input_df.columns = ['index','labels']

    data, labels_counts_sen = self.hoc_sentence2doc(input_df)

    print_log('HoC Dataset Details')
    print_log('No. of Documents:',len(data))
    print_log('No. of Sentences:',len(input_df))
    # print(labels_counts_sen)

    y_np, labels_counts_doc = self.hoc_labels2np(data)
    # print(y_np)
    # print(labels_counts_doc)
    res = pd.DataFrame()
    for lab in self.LABELS_HOC_FULL:
      print_log(lab,sum(labels_counts_doc[lab]),len(labels_counts_doc[lab]))
```

```
            temp_df = pd.DataFrame([[sum(labels_counts_doc[lab]),len(labels_counts_doc[lab])]], columns=['sum','len'], index=[lab])
            res = res.append(temp_df)

        return res

    @classmethod
    def eval_hoc(self, input_df):
        print_log("eval hoc",log_type="Function")

        # Reformat to Paper evaluator format
        ## Label need to be in a format of list of 10 cancers in fixed order
        true_df = input_df[['filename_line','label']].copy()
        pred_df = input_df[['filename_line','prediction']].copy()
        true_df.columns = ['index','labels']
        pred_df.columns = ['index','labels']

        # Group sentence back into documents
        data_true, true_labels_count_sen = self.hoc_sentence2doc(true_df)
        data_pred, pred_labels_count_sen = self.hoc_sentence2doc(pred_df)

        # merge data_true/pred into format of {'key':(set(true),set(pred))}
        assert data_true.keys() == data_pred.keys(), 'Key mismatch'
        all_keys = set(data_true.keys()).union(data_pred.keys())

        data = {}
        for k in all_keys:
            data[k] = (data_true[k],data_pred[k])
        # print(data)
        assert len(data) == 371, 'There are 371 documents in the test set: %d' % len(data)

        print_log('HoC Dataset Details')
        print_log('No. of Documents:',len(data))
        print_log('No. of Sentences:',len(true_df),'/',len(pred_df))
        print(true_labels_count_sen,pred_labels_count_sen)

        # Write into dataframe
        res_count_sen = pd.DataFrame()
        for lab in self.LABELS_HOC_FULL:
            temp_df = pd.DataFrame([[true_labels_count_sen[lab],pred_labels_count_sen[lab],len(true_df)]],
columns=['sentence_count_label','sentence_count_pred','sentence_count_total'], index=[lab])
            res_count_sen = res_count_sen.append(temp_df)
        # print(res_count)

        y_test, true_labels_count_doc = self.hoc_labels2np(data_true)
        y_pred, pred_labels_count_doc = self.hoc_labels2np(data_pred)

        res_count_doc = pd.DataFrame()
        for lab in self.LABELS_HOC_FULL:
            temp_df =
pd.DataFrame([[sum(true_labels_count_doc[lab]),sum(pred_labels_count_doc[lab]),len(true_labels_count_doc[lab])]],
columns=['doc_count_label','doc_count_pred','doc_count_total'], index=[lab])
            res_count_doc = res_count_doc.append(temp_df)
        res_confmat = pd.DataFrame(columns=['tn','fp','fn','tp'])

        # print(true_labels_list,pred_labels_list)
        for lab in self.LABELS_HOC_FULL:
            # print(lab)
            df2 = pd.DataFrame([list(confusion_matrix(true_labels_count_doc[lab],pred_labels_count_doc[lab]).ravel())],
columns=['tn','fp','fn','tp'], index=[lab])
            res_confmat = res_confmat.append(df2)
        # print(res_confmat)
        df_res = pd.concat([res_confmat,res_count_sen,res_count_doc], axis=1)
        print(df_res)

        r, p, f1 = self.get_p_r_f_arrary(y_pred, y_test)
        print('Precision: {:.6f}'.format(p))
        print('Recall   : {:.6f}'.format(r))
        print('F1       : {:.6f}'.format(f1))
        return float(r), float(p), float(f1) , df_res

    @classmethod
    def multi_acc(self, y_pred, y_test):
        y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
        _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)
```

```
        correct_pred = (y_pred_tags == y_test).float()
        acc = correct_pred.sum() / len(correct_pred)

        acc = torch.round(acc * 100)

        return acc

    @classmethod
    def eval_pubmedqa(self, input_df):
        class2idx = {
            'no':0,
            'maybe':1,
            'yes':2
        }
        idx2class = {v: k for k, v in class2idx.items()}
        input_df['label'].replace(idx2class, inplace=True)
        input_df['prediction'].replace(idx2class, inplace=True)

        confusion_matrix_df = pd.DataFrame(confusion_matrix(input_df.label.values, input_df.prediction.values))
        class_report = classification_report(input_df.label.values, input_df.prediction.values, digits=4,output_dict = True)
        print(class_report)
        # sns.heatmap(confusion_matrix_df, annot=True)
        return class_report , confusion_matrix_df

    @classmethod
    def eval_bioasq(self, input_df):
        class2idx = {
            'no':0,
            'yes':1
        }
        idx2class = {v: k for k, v in class2idx.items()}
        input_df['label'].replace(idx2class, inplace=True)
        input_df['prediction'].replace(idx2class, inplace=True)

        confusion_matrix_df = pd.DataFrame(confusion_matrix(input_df.label.values, input_df.prediction.values))
        class_report = classification_report(input_df.label.values, input_df.prediction.values, digits = 4,output_dict = True)
        print(class_report)
        sns.heatmap(confusion_matrix_df, annot=True)


        acc = accuracy_score(input_df.label.values, input_df.prediction.values)
        prc,rec,f_score,_ = precision_recall_fscore_support(input_df.label.values, input_df.prediction.values, average='macro')

        print("acc",acc)
        print("prc",prc)
        print("rec",rec)
        print("f_score",f_score)

        return class_report , confusion_matrix_df

# eval_hoc(temp_df)

# temp_df = bert_test.df_BERT.copy()
# temp_df['prediction'] = pd.Series(map(lambda x: [str(i)+"_"+str(int(x[i])) for i in range(len(x))], predict_y))
# temp_df['prediction'] = temp_df['prediction'].apply(lambda x: ','.join(x))
# r, p, f1 = my_evaluator.eval_hoc(temp_df)

"""---

## Model Generator

### Fine-tuned Hall-of-Cancer (HoC)

#### Assisting function
"""

def text_dependent(df_input, shift_level=0):
    temp_df = df_input.copy()
    new = temp_df['filename_line'].str.split("_", n = 1, expand = True)
    # making separate first name column from new data frame
    temp_df["filename"]= new[0]
    # making separate last name column from new data frame
    temp_df["sentence"]= new[1]
```

```
    if(shift_level==1):
        df_input['text'] = temp_df.groupby('filename').text.apply(lambda x: x.shift(1).fillna('')+' '+ x).str.strip()
    elif(shift_level==2):
        df_input['text'] = temp_df.groupby('filename').text.apply(lambda x: x.shift(2).fillna('')+' '+ x.shift(1).fillna('')+' '+ x).str.strip()
    elif(shift_level==3):
        df_input['text'] = temp_df.groupby('filename').text.apply(lambda x: x.shift(3).fillna('')+' '+ x.shift(2).fillna('')+' '+
x.shift(1).fillna('')+' '+ x).str.strip()
    return df_input

def transform_dataset(DATAPATH,DATASET,PRETRAIN_MODEL='bert-base-uncased',TOKEN_SIZE=128, SHIFT_LEVEL=None,
FORCE=False):
    print_log("Try loading data from cache")
    temp_path = os.path.join(DATAPATH,"datasets","transformed",DATASET,PRETRAIN_MODEL)
    if(SHIFT_LEVEL == None or SHIFT_LEVEL == 0):
        temp_path = os.path.join(temp_path,"token_length_"+str(TOKEN_SIZE))
    else:
        temp_path = os.path.join(temp_path,"token_length_"+str(TOKEN_SIZE)+"_shift_"+str(SHIFT_LEVEL))

    temppath_train = os.path.join(temp_path,"train.csv")
    temppath_valid = os.path.join(temp_path,"valid.csv")
    temppath_test = os.path.join(temp_path,"test.csv")

    print_log("Train file exist:",os.path.isfile(temppath_train),"(",temppath_train,")")
    print_log("Valid file exist:",os.path.isfile(temppath_valid),"(",temppath_valid,")")
    print_log("Test file exist:",os.path.isfile(temppath_test),"(",temppath_test,")")

    if((os.path.isfile(temppath_train) and os.path.isfile(temppath_valid) and os.path.isfile(temppath_valid)) and not FORCE):
        df_train = pd.read_csv(temppath_train).iloc[:, 1:]
        df_valid = pd.read_csv(temppath_valid).iloc[:, 1:]
        df_test = pd.read_csv(temppath_test).iloc[:, 1:]
        bert_train = my_BERT(df_train, is_transform=True)
        bert_test = my_BERT(df_test, is_transform=True)
        bert_valid = my_BERT(df_valid, is_transform=True)
        print_log("Load Data From Existing",log_type="Success")
        return bert_train, bert_test, bert_valid

    print_log("Transform Data","FORCE" if FORCE else "")
    temp_path = os.path.join(DATAPATH,"datasets","raw",DATASET)
    temppath_train = os.path.join(temp_path,"train.tsv")
    temppath_valid = os.path.join(temp_path,"dev.tsv")
    temppath_test = os.path.join(temp_path,"test.tsv")

    df_train = pd.read_csv(temppath_train, sep='\t')
    df_test = pd.read_csv(temppath_test, sep='\t')
    df_valid = pd.read_csv(temppath_valid, sep='\t')

    # TO DO : Modify this if not HoC
    df_train.columns = ['label','text','filename_line']
    df_test.columns = ['label','text','filename_line']
    df_valid.columns = ['label','text','filename_line']

    if(SHIFT_LEVEL != None):
        df_train = text_dependent(df_train,SHIFT_LEVEL)
        df_test = text_dependent(df_test,SHIFT_LEVEL)
        df_valid = text_dependent(df_valid,SHIFT_LEVEL)

    bert_train = my_BERT(df_train)
    bert_test = my_BERT(df_test)
    bert_valid = my_BERT(df_valid)

    bert_train.load_pretrain_bert(PRETRAIN_MODEL)
    bert_test.load_pretrain_bert(PRETRAIN_MODEL)
    bert_valid.load_pretrain_bert(PRETRAIN_MODEL)

    print_log("BERTTransform: Train Data")
    bert_train.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)
    print_log("BERTTransform: Test Data")
    bert_test.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)
    print_log("BERTTransform: Valid Data")
    bert_valid.bert_transform(DEVICE_AVAILABLE, token_length=TOKEN_SIZE)

    if(SHIFT_LEVEL == None):
        temp_path =
os.path.join(DATAPATH,"datasets","transformed",DATASET,PRETRAIN_MODEL,"token_length_"+str(TOKEN_SIZE))
```

```
    else:
        temp_path =
os.path.join(DATAPATH,"datasets","transformed",DATASET,PRETRAIN_MODEL,"token_length_"+str(TOKEN_SIZE)+"_shift_"+str(
SHIFT_LEVEL))
    Path(os.path.join(temp_path)).mkdir(parents=True, exist_ok=True)

    temp_df = bert_train.get_features_df(['filename_line','label'])
    temp_df.to_csv(os.path.join(temp_path,"train.csv"))
    print_log(len(df_train),"/",len(temp_df))
    temp_df = bert_test.get_features_df(['filename_line','label'])
    temp_df.to_csv(os.path.join(temp_path,"test.csv"))
    print_log(len(df_test),"/",len(temp_df))
    temp_df = bert_valid.get_features_df(['filename_line','label'])
    temp_df.to_csv(os.path.join(temp_path,"valid.csv"))
    print_log(len(df_valid),"/",len(temp_df))

    return bert_train, bert_test, bert_valid

# transform_dataset(DATAPATH = "/content/drive/MyDrive/MinorThesis/",
#     DATASET = "HoC",
#     TOKEN_SIZE = 128,
#     PRETRAIN_MODEL = 'biobert-base-uncased',
#     SHIFT_LEVEL = None,
#     FORCE = True)

def train_linear_model(BERT_TRAIN, BERT_TEST, BERT_VALID, LABELS=['label'], LEARNING_RATE = 1e-5, BATCH_SIZE =
32, N_ITER=3000):
    print_log("learning_rate:",LEARNING_RATE)
    print_log("batch_size",BATCH_SIZE)
    print_log("N_ITER",N_ITER)
    print_log("Training Downstream Model")

    model = my_downstream(BERT_TRAIN.get_features(),BERT_TRAIN.get_labels(LABELS),
                        BERT_TEST.get_features(),BERT_TEST.get_labels(LABELS),
                        BERT_VALID.get_features(),BERT_VALID.get_labels(LABELS))
    train_loss, valid_loss = model.train_linear_nn(D_in = 768,
                        n_classes = 1 if type(LABELS) == str else len(LABELS),
                        n_iter=N_ITER,
                        learning_rate = LEARNING_RATE,
                        batch_size = BATCH_SIZE)
    return model, train_loss, valid_loss

"""#### Execution"""

# HoC
def run_dsgenerator(DATAPATH, DATASET, LABELS=['label'], PRETRAIN_MODEL = 'bert-base-uncased', TOKEN_SIZE = 128,
SHIFT_LEVEL = None , LEARNING_RATE = 1e-5, BATCH_SIZE = 32, N_ITER=3000):
    print_log("Run Generator","Function")

    # UNIQUE IDENTIFIER USING
    sttime = datetime.now().strftime('%Y%m%d_%H-%M-%S')

    bert_train, bert_test, bert_valid = transform_dataset(DATAPATH = DATAPATH,
                                            DATASET = DATASET,
                                            PRETRAIN_MODEL = PRETRAIN_MODEL,
                                            SHIFT_LEVEL = SHIFT_LEVEL,
                                            TOKEN_SIZE = TOKEN_SIZE)

    bert_train.extract_hoc_label()
    bert_test.extract_hoc_label()
    bert_valid.extract_hoc_label()

    # TRAINING
    model, train_loss, valid_loss = train_linear_model(bert_train, bert_test, bert_valid,
                            LABELS, LEARNING_RATE, BATCH_SIZE, N_ITER)
    # print_log("learning_rate:",LEARNING_RATE)
    # print_log("batch_size",BATCH_SIZE)
    # print_log("Training Downstream Model")
    # model = my_downstream(bert_train.get_features(),bert_train.get_labels(LABELS),
    #                     bert_test.get_features(),bert_test.get_labels(LABELS),
    #                     bert_valid.get_features(),bert_valid.get_labels(LABELS))
    # train_loss, valid_loss = model.train_linear_nn(D_in = 768,
    #                     n_classes = 1 if type(LABELS) == str else len(LABELS),
    #                     n_iter=N_ITER,
```

```
    #                   learning_rate = LEARNING_RATE,
    #                   batch_size = BATCH_SIZE)

    _,predict_y = model.predict()

    temp_df = bert_test.df_BERT.copy()
    temp_df['prediction'] = pd.Series(map(lambda x: [str(i)+"_"+str(int(x[i])) for i in range(len(x))], predict_y))
    temp_df['prediction'] = temp_df['prediction'].apply(lambda x: ','.join(x))
    r, p, f1, _ = my_evaluator.eval_hoc(temp_df)

    result = {}
    result['dataset'] = DATASET
    result['labels'] = LABELS

    result['pretrain_model'] = PRETRAIN_MODEL + '-tks' + str(TOKEN_SIZE)
    result['downstream_model'] = model.get_model_name()
    result['downstream_model_savepoint'] = "model_"+sttime+".pt"

    result['recall'] = r
    result['precision'] = p
    result['f1score'] = f1

    result['best_iter'] = model.best_iter

    result['SHIFT_LEVEL'] = SHIFT_LEVEL
    result['LEARNING_RATE'] = LEARNING_RATE
    result['BATCH_SIZE'] = BATCH_SIZE

    # result['hyper_param'] = hp
    # result['predict_y'] = predict_y

    # SAVING SECTION
    temp_path_model = os.path.join(DATAPATH,"models",DATASET,PRETRAIN_MODEL)
    temp_path_result = os.path.join(DATAPATH,"results",DATASET,PRETRAIN_MODEL)

    Path(temp_path_model).mkdir(parents=True, exist_ok=True)
    Path(temp_path_result).mkdir(parents=True, exist_ok=True)

    # SAVE MODEL
    torch.save(model.best_model.state_dict(), os.path.join(temp_path_model,"model_"+sttime+".pt"))

    res_df = pd.DataFrame(result.items(), columns=['key', 'result']).set_index('key')
    res_df.to_json(os.path.join(temp_path_result,"result_"+sttime+".json"))

    # SAVE PLOT
    plt.plot(train_loss, label="train")
    plt.plot(valid_loss, label="valid")
    plt.title(PRETRAIN_MODEL+"-"+DATASET+"-"+str(TOKEN_SIZE))
    plt.suptitle(str(BATCH_SIZE)+"-"+str(LEARNING_RATE))
    plt.ylabel('loss')
    plt.legend()
    plt.savefig(os.path.join(temp_path_result,"result_"+sttime+".png"))

    print_log("",log_type="----------")
    # return predict_y
    return result

# list_cancer = ['label_IM', 'label_ID', 'label_CE', 'label_RI', 'label_GS', 'label_GI', 'label_A', 'label_CD', 'label_PS', 'label_TPI']

# TARGET_LABEL = list_cancer
# TARGET_DATASET = "HoC" # "dat_hoc","dat_semi"
# TARGET_PATH = '/content/drive/MyDrive/MinorThesis/'
# PRETRAIN_MODEL = 'bert-base-uncased'
# N_ITER=1000

# BATCH_SIZE = 16
# LEARNING_RATE = 5e-5
# SHIFT_LEVEL=None

# # for i in range(10):
# res = run_dsgenerator(TARGET_PATH,TARGET_DATASET,TARGET_LABEL,
#                       TOKEN_SIZE=512, PRETRAIN_MODEL=PRETRAIN_MODEL,
#                       LEARNING_RATE = LEARNING_RATE, BATCH_SIZE = BATCH_SIZE,
#                       N_ITER = N_ITER, SHIFT_LEVEL=SHIFT_LEVEL)
```

```
# res

# # # For test in Colab Only
# DATAPATH = "/content/drive/MyDrive/MinorThesis/"
# DATASET = "HoC"
# TOKEN_SIZE = 128
# PRETRAIN_MODEL = 'bert-base-uncased'
# SHIFT_LEVEL = None
# IS_TRANSFORM = True

# temp_path = os.path.join(DATAPATH,"datasets","transformed",DATASET,PRETRAIN_MODEL);
# if(SHIFT_LEVEL == None):
#     temp_path = os.path.join(temp_path,"token_length_"+str(TOKEN_SIZE))
# else:
#     temp_path = os.path.join(temp_path,"token_length_"+str(TOKEN_SIZE)+"_shift_"+str(SHIFT_LEVEL))

# temppath_train = os.path.join(temp_path,"train.csv")
# temppath_valid = os.path.join(temp_path,"valid.csv")
# temppath_test = os.path.join(temp_path,"test.csv")

# print_log("Train file exist:",os.path.isfile(temppath_train),"(",temppath_train,")")
# print_log("Valid file exist:",os.path.isfile(temppath_valid),"(",temppath_valid,")")
# print_log("Test file exist:",os.path.isfile(temppath_test),"(",temppath_test,")")

# df_train = pd.read_csv(temppath_train).iloc[:, 1:]
# df_valid = pd.read_csv(temppath_valid).iloc[:, 1:]
# df_test = pd.read_csv(temppath_test).iloc[:, 1:]

# bert_train = my_BERT(extract_hoc_label(df_train),is_transform=IS_TRANSFORM)
# bert_valid = my_BERT(extract_hoc_label(df_valid),is_transform=IS_TRANSFORM)
# bert_test = my_BERT(extract_hoc_label(df_test),is_transform=IS_TRANSFORM)

# BATCH_SIZE = 32
# LEARNING_RATE = 5e-5
# SHIFT_LEVEL=3

# res = run_dsgenerator(TARGET_PATH,TARGET_DATASET,TARGET_LABEL,
#                   TOKEN_SIZE=512, PRETRAIN_MODEL=PRETRAIN_MODEL,
#                   LEARNING_RATE = LEARNING_RATE, BATCH_SIZE = BATCH_SIZE,
#                   N_ITER = N_ITER, SHIFT_LEVEL=SHIFT_LEVEL)
# res

"""### Fine-tuned PubMedQA

#### Assisting function
"""

# Extract Fold ID (test,train)

def get_pubmedqa_fold_id(DATAPATH,DATASET):
    list_data_fold = []
    for i in range(10):
        temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"train_set.json")
        temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"dev_set.json")

        df_temp_train = pd.read_json(temppath_train).transpose().reset_index()
        df_temp_valid = pd.read_json(temppath_valid).transpose().reset_index()

        list_data_fold.append((df_temp_train['index'].values,df_temp_valid['index'].values))
        # list_data_fold.append((df_temp_train,df_temp_valid))
        # print(df_temp_train.shape,df_temp_valid.shape)
    return list_data_fold

def get_class_distribution(obj):
    count_dict = {
        "rating_no": 0,
        "rating_maybe": 0,
        "rating_yes": 0,
    }

    for i in obj:
        if i == 0:
            count_dict['rating_no'] += 1
        elif i == 1:
```

```
            count_dict['rating_maybe'] += 1
        elif i == 2:
            count_dict['rating_yes'] += 1
        else:
            print("Check classes.")

    return count_dict

def pubmedqa_train_linear_model(BERT_TRAIN, BERT_TEST, FOLDS_IDX, FOLD_I=0, LABELS=['label'], LEARNING_RATE =
1e-5, BATCH_SIZE = 32, N_ITER=3000):
    print_log("learning_rate:",LEARNING_RATE)
    print_log("batch_size",BATCH_SIZE)
    print_log("N_ITER",N_ITER)
    print_log("Training Downstream Model")

    df_temp = BERT_TRAIN.df_BERT.copy()
    df_train = df_temp[df_temp.id.isin(FOLDS_IDX[FOLD_I][0])]
    df_valid = df_temp[df_temp.id.isin(FOLDS_IDX[FOLD_I][1])]

    bert_train = my_BERT(df_train, is_transform=True)
    bert_valid = my_BERT(df_valid, is_transform=True)

    # print(np.array(df_train['label']))
    # fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(25,7))
    # # Train
    # sns.barplot(data = pd.DataFrame.from_dict([get_class_distribution(np.array(df_train['label']))]).melt(), x = "variable", y="value",
hue="variable",  ax=axes[0]).set_title('Class Distribution in Train Set')
    # # Validation
    # sns.barplot(data = pd.DataFrame.from_dict([get_class_distribution(np.array(df_valid['label']))]).melt(), x = "variable", y="value",
hue="variable",  ax=axes[1]).set_title('Class Distribution in Val Set')
    # # # Test
    # # sns.barplot(data = pd.DataFrame.from_dict([get_class_distribution(y_test)]).melt(), x = "variable", y="value", hue="variable",
ax=axes[2]).set_title('Class Distribution in Test Set')

    model = my_downstream(bert_train.get_features(),bert_train.get_labels(LABELS),
                          BERT_TEST.get_features(),BERT_TEST.get_labels(LABELS),
                          bert_valid.get_features(),bert_valid.get_labels(LABELS))
    model.train_multiclass_nn(D_in = 768,
                              n_classes = 3,
                              EPOCHS=N_ITER,
                              learning_rate = LEARNING_RATE,
                              batch_size = BATCH_SIZE)

    # Create dataframes
    train_val_acc_df =
pd.DataFrame.from_dict(model.accuracy_stats).reset_index().melt(id_vars=['index']).rename(columns={"index":"epochs"})
    train_val_loss_df =
pd.DataFrame.from_dict(model.loss_stats).reset_index().melt(id_vars=['index']).rename(columns={"index":"epochs"})
    # Plot the dataframes
    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(20,7))
    sns.lineplot(data=train_val_acc_df, x = "epochs", y="value", hue="variable",  ax=axes[0]).set_title('Train-Val Accuracy/Epoch')
    sns.lineplot(data=train_val_loss_df, x = "epochs", y="value", hue="variable", ax=axes[1]).set_title('Train-Val Loss/Epoch')

    return model

def pubmedqa_transform_dataset(DATAPATH,DATASET,PRETRAIN_MODEL='bert-base-uncased',TOKEN_SIZE=128,
SHIFT_LEVEL=None, FORCE=False, REASONING=False):
    print_log("Try loading data from cache")
    temp_path = os.path.join(DATAPATH,"datasets","transformed",DATASET,"QuesAbs", "reasoning_required" if REASONING else
"reasoning_free", PRETRAIN_MODEL)
    if(SHIFT_LEVEL == None):
        temp_path = os.path.join(temp_path,"token_length_"+str(TOKEN_SIZE))
    else:
        temp_path = os.path.join(temp_path,"token_length_"+str(TOKEN_SIZE)+"_shift_"+str(SHIFT_LEVEL))

    temppath_train = os.path.join(temp_path,"train.csv")
    temppath_test = os.path.join(temp_path,"test.csv")

    print_log("Train file exist:",os.path.isfile(temppath_train),"(",temppath_train,")")
    print_log("Test file exist:",os.path.isfile(temppath_test),"(",temppath_test,")")

    if((os.path.isfile(temppath_train) and os.path.isfile(temppath_test)) and not FORCE):
        df_train = pd.read_csv(temppath_train).iloc[:, 1:]
        df_test = pd.read_csv(temppath_test).iloc[:, 1:]
```

```
        bert_train = my_BERT(df_train, is_transform=True)
        bert_test = my_BERT(df_test, is_transform=True)
        print_log("Load Data From Existing",log_type="Success")
        return bert_train, bert_test

    print_log("Transform Data","FORCE" if FORCE else "")
    ### COPY FROM TRANSFOR DATASET FOR PUBMEDQA
    pass

# Pubmedqa
def run_pubmedqa_dsgenerator_allfold(DATAPATH, DATASET, LABELS=['label'], PRETRAIN_MODEL = 'bert-base-uncased',
TOKEN_SIZE = 128, SHIFT_LEVEL = None , LEARNING_RATE = 1e-5, BATCH_SIZE = 32, N_ITER=3000, REASONING=False):
    print_log("Run Generator","Function")
    # UNIQUE IDENTIFIER USING
    sttime = datetime.now().strftime('%Y%m%d_%H-%M-%S')



    ####
    print_log("PRETRAIN_MODEL: ",PRETRAIN_MODEL)
    print_log("REASONING: ",REASONING )
    bert_train, bert_test = pubmedqa_transform_dataset(DATAPATH = DATAPATH,
                                            DATASET = DATASET,
                                            PRETRAIN_MODEL = PRETRAIN_MODEL,
                                            SHIFT_LEVEL = SHIFT_LEVEL,
                                            TOKEN_SIZE = TOKEN_SIZE,
                                          REASONING = REASONING)

    bert_train.class2idx_pubmedqa_label()
    bert_test.class2idx_pubmedqa_label()

    folds_idx = get_pubmedqa_fold_id(DATAPATH, DATASET)

    list_acc = []
    temp_df = bert_test.df_BERT.copy()

    for i in range(10):
        print_log("######## FOLD: ",i)
        # TRAINING
        model = pubmedqa_train_linear_model(bert_train, bert_test, folds_idx, i,
                            LABELS, LEARNING_RATE, BATCH_SIZE, N_ITER)

        # TEST
        _,predict_y = model.predict()

        temp_df['prediction'] = pd.Series(predict_y)
        class_report, confusion_matrix_df = my_evaluator.eval_pubmedqa(temp_df)
        temp_df.rename(columns={'prediction': 'reason_'+str(REASONING)+'_fold'+str(i)},inplace=True)
        list_acc.append(class_report['accuracy'])

        # SAVING SECTION
        temp_path_model = os.path.join(DATAPATH,"models",DATASET,PRETRAIN_MODEL,'reason_'+str(REASONING),sttime)
        Path(temp_path_model).mkdir(parents=True, exist_ok=True)

        # SAVE MODEL
        torch.save(model.best_model.state_dict(), os.path.join(temp_path_model,"model_"+"fold_"+str(i)+".pt"))

    ####
    print_log("PRETRAIN_MODEL: ",PRETRAIN_MODEL)
    print_log("REASONING: ",not REASONING )
    bert_train, bert_test = pubmedqa_transform_dataset(DATAPATH = DATAPATH,
                                            DATASET = DATASET,
                                            PRETRAIN_MODEL = PRETRAIN_MODEL,
                                            SHIFT_LEVEL = SHIFT_LEVEL,
                                            TOKEN_SIZE = TOKEN_SIZE,
                                          REASONING = not REASONING)

    # bert_train.class2idx_pubmedqa_label()
    # bert_test.class2idx_pubmedqa_label()

    # folds_idx = get_pubmedqa_fold_id(DATAPATH, DATASET)
    # for i in range(10):
    #     # TRAINING
    #     print(i)
```

```
    #     model = pubmedqa_train_linear_model(bert_train, bert_test, folds_idx, i,
    #                                 LABELS, LEARNING_RATE, BATCH_SIZE, N_ITER)
    #     # TEST
    #     _,predict_y = model.predict()

    #     temp_df['prediction'] = pd.Series(predict_y)
    #     class_report, confusion_matrix_df = my_evaluator.eval_pubmedqa(temp_df)
    #     temp_df.rename(columns={'prediction': 'reason_'+str(not REASONING)+'_fold'+str(i)},inplace=True)
    #     list_acc.append(class_report['accuracy'])

    #     # SAVING SECTION
    #     temp_path_model = os.path.join(DATAPATH,"models",DATASET,PRETRAIN_MODEL,'reason_'+str(not REASONING),sttime)
    #     Path(temp_path_model).mkdir(parents=True, exist_ok=True)

    #     # SAVE MODEL
    #     torch.save(model.best_model.state_dict(), os.path.join(temp_path_model,"model_"+"fold_"+str(i)+".pt"))

    # #### MERGE
    # print_log("REASONING MERGE")
    # temp_df['prediction'] = temp_df.apply(lambda x : x['reason_True'] if x['reason_True']==x['reason_False'] else "maybe", axis=1)
    # class_report, confusion_matrix_df = my_evaluator.eval_pubmedqa(temp_df)

    # result = {}
    # result['dataset'] = DATASET
    # result['labels'] = LABELS

    # result['pretrain_model'] = PRETRAIN_MODEL + '-tks' + str(TOKEN_SIZE)
    # result['downstream_model'] = model.get_model_name()
    # result['downstream_model_savepoint'] = "model_"+sttime+".pt"

    # result['summary'] = class_report
    # # result['recall'] = r
    # # result['precision'] = p
    # # result['f1score'] = f1

    # result['best_iter'] = model.best_iter

    # result['LEARNING_RATE'] = LEARNING_RATE
    # result['BATCH_SIZE'] = BATCH_SIZE

    # # SAVING SECTION
    # temp_path_model = os.path.join(DATAPATH,"models",DATASET,"reasoning_required" if REASONING else
"reasoning_free",PRETRAIN_MODEL)
    # temp_path_result = os.path.join(DATAPATH,"results",DATASET,"reasoning_required" if REASONING else
"reasoning_free",PRETRAIN_MODEL)

    # Path(temp_path_model).mkdir(parents=True, exist_ok=True)
    # Path(temp_path_result).mkdir(parents=True, exist_ok=True)

    # # SAVE MODEL
    # torch.save(model.best_model.state_dict(), os.path.join(temp_path_model,"model_"+sttime+".pt"))

    # res_df = pd.DataFrame(result.items(), columns=['key', 'result']).set_index('key')
    # res_df.to_json(os.path.join(temp_path_result,"result_"+sttime+".json"))

    arr = np.array(list_acc)
    arr2 = arr.reshape((1,10))
    print_log("Results (REASONING_REQUIRED, REASONING_FREE):",np.average(arr2, axis=1),log_type="SUCCESS")

    return list_acc

TARGET_LABEL = ['label']
TARGET_DATASET = "pubmedqa"
TARGET_PATH = '/content/drive/MyDrive/MinorThesis/'
PRETRAIN_MODEL = 'biobert-base-cased'
N_ITER=300

BATCH_SIZE = 32
LEARNING_RATE = 5e-5
SHIFT_LEVEL=None

# for bs in [16, 32]:
#     for lr in [1e-5, 3e-3, 5e-5]:
```

```
# for bert_model in ['biobert-base-cased']:
#     for bs in [32]:
#         for lr in [3e-3]:
#             for i in range(45):
#                 res = run_pubmedqa_dsgenerator_allfold(TARGET_PATH,TARGET_DATASET,TARGET_LABEL,
#                                 TOKEN_SIZE=512, PRETRAIN_MODEL=bert_model,
#                                 LEARNING_RATE = lr, BATCH_SIZE = bs,
#                                 N_ITER = N_ITER, SHIFT_LEVEL=SHIFT_LEVEL,
#                                 REASONING = True)
# res


# res = run_pubmedqa_dsgenerator_allfold(TARGET_PATH,TARGET_DATASET,TARGET_LABEL,
#                     TOKEN_SIZE=512, PRETRAIN_MODEL=PRETRAIN_MODEL,
#                     LEARNING_RATE = LEARNING_RATE, BATCH_SIZE = BATCH_SIZE,
#                     N_ITER = N_ITER, SHIFT_LEVEL=SHIFT_LEVEL,
#                     REASONING = True)
# res

def run_pubmedqa_ensemble(DATAPATH, DATASET, MODEL_GROUP, LABELS=['label'], TOKEN_SIZE=128, SHIFT_LEVEL =
None):
    print_log("Run Ensemble","Function")

    temp_path = os.path.join(DATAPATH,"models",DATASET,"ensemble",MODEL_GROUP)
    LIST_PRETRAIN_MODEL = os.listdir(temp_path)
    print_log("Ensemble pretrain model list:",LIST_PRETRAIN_MODEL)

    list_acc = []
    list_results = []
    list_results_f1 = []

    all_pred_df = pd.DataFrame()

    for pretrain_model in LIST_PRETRAIN_MODEL:
        print_log("model:",pretrain_model)

        print_log("PRETRAIN_MODEL: ",pretrain_model)
        REASONING = True
        print_log("REASONING: ",REASONING)
        bert_train, bert_test = pubmedqa_transform_dataset(DATAPATH = DATAPATH,
                                            DATASET = DATASET,
                                            PRETRAIN_MODEL = pretrain_model,
                                            SHIFT_LEVEL = SHIFT_LEVEL,
                                            TOKEN_SIZE = TOKEN_SIZE,
                                            REASONING = REASONING)

        bert_train.class2idx_pubmedqa_label()
        bert_test.class2idx_pubmedqa_label()

        folds_idx = get_pubmedqa_fold_id(DATAPATH, DATASET)

        model = my_downstream(bert_train.get_features(),bert_train.get_labels(LABELS), # Not going to be used
                        bert_test.get_features(),bert_test.get_labels(LABELS), # Not going to be used
                        bert_train.get_features(),bert_train.get_labels(LABELS))

        temp_path2 = os.path.join(temp_path,pretrain_model,"reason_True")
        LIST_MODEL = os.listdir(temp_path2)

        print()
        print()
        print_log("Ensemble pretrain LIST_MODEL:",LIST_MODEL)
        for group_model in LIST_MODEL:
            temp_df = bert_test.df_BERT.copy()
            temp_path3 = os.path.join(temp_path2,group_model)
            LIST_FOLDS = os.listdir(temp_path3)

            print()
            print()
            print_log("Ensemble pretrain LIST_FOLDS:",str(len(LIST_FOLDS)),LIST_FOLDS)
            for downsteam_model in LIST_FOLDS:
                print_log("FULL PATH:",temp_path3,downsteam_model)
                model.load_multiclass_nn(D_in = 768,
                                n_classes = 3,
```

```
                        model_path = os.path.join(temp_path3,downsteam_model))

            # print(model)
            # TEST
            _,predict_y = model.predict()
            list_results.append(predict_y)

            all_pred_df[group_model+"_"+downsteam_model] = pd.Series(predict_y)
            temp_df['prediction'] = pd.Series(predict_y)
            class_report, confusion_matrix_df = my_evaluator.eval_pubmedqa(temp_df)
            temp_df.rename(columns={'prediction': group_model+"_"+downsteam_model},inplace=True)
            list_acc.append(class_report['accuracy'])



    majority_df = all_pred_df.apply(pd.Series.value_counts, axis=1).fillna(0)
    # majority_df
    maxValuesObj = majority_df.idxmax(axis=1)
    maxValuesObj
    print()
    print()

    print_log("Final Results of",len(list_results),"models",log_type='Success')
    # ensemble_res = sum(list_results)
    # ensemble_res = ensemble_res/np.amax(ensemble_res) # Majority Vote
    # # ensemble_res[ensemble_res>=1] = 1 # Only 1 is positive

    temp_df = bert_test.df_BERT.copy()
    temp_df['prediction'] = maxValuesObj
    class_report, confusion_matrix_df = my_evaluator.eval_pubmedqa(temp_df)
    print(confusion_matrix_df)
    print(class_report)

    return list_acc,all_pred_df

"""#### Execution"""

TARGET_LABEL = ['label']
TARGET_DATASET = "pubmedqa" # "dat_hoc","dat_semi"
TARGET_PATH = '/content/drive/MyDrive/MinorThesis/'

MODEL_GROUP = "1_pubmedbert"

list_acc,temp_df = run_pubmedqa_ensemble(TARGET_PATH,TARGET_DATASET,
                        MODEL_GROUP = MODEL_GROUP,
                        LABELS = TARGET_LABEL,
                        TOKEN_SIZE=512)

temp_df

"""### Fine-tuned BioASQ

#### Assisting function
"""

# HoC
def run_bioasq_dsgenerator(DATAPATH, DATASET, LABELS=['label'], PRETRAIN_MODEL = 'bert-base-uncased', TOKEN_SIZE =
128, SHIFT_LEVEL = None , LEARNING_RATE = 1e-5, BATCH_SIZE = 32, N_ITER=3000):
    print_log("Run Generator","Function")

    # UNIQUE IDENTIFIER USING
    sttime = datetime.now().strftime('%Y%m%d_%H-%M-%S')

    bert_train, bert_test, bert_valid = transform_dataset(DATAPATH = DATAPATH,
                                            DATASET = DATASET,
                                            PRETRAIN_MODEL = PRETRAIN_MODEL,
                                            SHIFT_LEVEL = SHIFT_LEVEL,
                                            TOKEN_SIZE = TOKEN_SIZE)
    bert_train.class2idx_bioasq_label()
    bert_valid.class2idx_bioasq_label()
    bert_test.class2idx_bioasq_label()


    # # TRAINING
```

```
    model, train_loss, valid_loss = train_linear_model(bert_train, bert_test, bert_valid,
                            LABELS, LEARNING_RATE, BATCH_SIZE, N_ITER)

    _,predict_y = model.predict()

    temp_df = bert_test.df_BERT.copy()
    temp_df['prediction'] = pd.Series(predict_y)
    class_report, confusion_matrix_df = my_evaluator.eval_bioasq(temp_df)

    # result = {}
    # result['dataset'] = DATASET
    # result['labels'] = LABELS

    # result['pretrain_model'] = PRETRAIN_MODEL + '-tks' + str(TOKEN_SIZE)
    # result['downstream_model'] = model.get_model_name()
    # result['downstream_model_savepoint'] = "model_"+sttime+".pt"

    # result['recall'] = r
    # result['precision'] = p
    # result['f1score'] = f1

    # result['best_iter'] = model.best_iter

    # result['SHIFT_LEVEL'] = SHIFT_LEVEL
    # result['LEARNING_RATE'] = LEARNING_RATE
    # result['BATCH_SIZE'] = BATCH_SIZE

    # # result['hyper_param'] = hp
    # # result['predict_y'] = predict_y

    # # SAVING SECTION
    # temp_path_model = os.path.join(DATAPATH,"models",DATASET,PRETRAIN_MODEL)
    # temp_path_result = os.path.join(DATAPATH,"results",DATASET,PRETRAIN_MODEL)

    # Path(temp_path_model).mkdir(parents=True, exist_ok=True)
    # Path(temp_path_result).mkdir(parents=True, exist_ok=True)

    # # SAVE MODEL
    # torch.save(model.best_model.state_dict(), os.path.join(temp_path_model,"model_"+sttime+".pt"))

    # res_df = pd.DataFrame(result.items(), columns=['key', 'result']).set_index('key')
    # res_df.to_json(os.path.join(temp_path_result,"result_"+sttime+".json"))

    # # SAVE PLOT
    # plt.plot(train_loss, label="train")
    # plt.plot(valid_loss, label="valid")
    # plt.title(PRETRAIN_MODEL+"-"+DATASET+"-"+str(TOKEN_SIZE))
    # plt.suptitle(str(BATCH_SIZE)+"-"+str(LEARNING_RATE))
    # plt.ylabel('loss')
    # plt.legend()
    # plt.savefig(os.path.join(temp_path_result,"result_"+sttime+".png"))

    # print_log("",log_type="----------")
    # # return predict_y
    # return result

"""#### Execution"""

TARGET_LABEL = 'label'
TARGET_DATASET = "BioASQ" # "dat_hoc","dat_semi"
TARGET_PATH = '/content/drive/MyDrive/MinorThesis/'
PRETRAIN_MODEL = 'biobert-base-cased'
N_ITER=1000

BATCH_SIZE = 32
LEARNING_RATE = 5e-5
SHIFT_LEVEL=None


res = run_bioasq_dsgenerator(TARGET_PATH,TARGET_DATASET,TARGET_LABEL,
                    TOKEN_SIZE=512, PRETRAIN_MODEL=PRETRAIN_MODEL,
                    LEARNING_RATE = LEARNING_RATE, BATCH_SIZE = BATCH_SIZE,
                    N_ITER = N_ITER, SHIFT_LEVEL=SHIFT_LEVEL)
res
```

```
res.df_BERT

"""## Model Results Read """

import os
import pandas as pd
pd.set_option('display.max_colwidth', 0)

TARGET_PATH = '/content/drive/MyDrive/MinorThesis/'

# temp_list = []
# datasets = os.listdir(TARGET_PATH+"results")
# print(datasets)
# for ds in datasets:
#     files = os.listdir(TARGET_PATH+"results/"+ds)
#     # print(arr2)
#     for f in files:
#         print(TARGET_PATH+"results/"+ds+"/"+f)
#         try:
#             a = pd.read_json(TARGET_PATH+"results/"+ds+"/"+f)
#             a = a.transpose()
#             a.index = [f]
#             temp_list.append(a)
#         except:
#             print("error")
#             pass
# temp_res = pd.concat(temp_list, sort=False)
# # temp_res.drop(columns='predict_y',inplace=True) # This cause error when display result
# # temp_res.drop_duplicates(keep='last',inplace=True)

temp_res = temp_res[['dataset','pretrain_model','hyper_param','f1score','precision','recall']]

# temp_res.drop(columns='predict_y',inplace=True)
# temp_res.drop(columns='labels',inplace=True)
temp_res.drop_duplicates(keep='last',inplace=True)

# Table 1 : Show comparison between different type of BERT
# ONLY Original BERT IS WORST!!!
temp_res[(temp_res.dataset=='blurb_hoc')].sort_values(['f1score'], ascending = False)
```

3. text_classification_on_blurb_hoc_traditional_fine_tuned.py

```
# -*- coding: utf-8 -*-
"""Text Classification on BLURB - HoC - Traditional Fine-tuned

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1yyPkt9loRTzsWalRPoS9miODNal7BacU
"""

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)

from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')

"""If you're opening this Notebook on colab, you will probably need to install 🤗 Transformers and 🤗 Datasets. Uncomment the following cell and run it."""

! pip install datasets transformers

"""If you're opening this notebook locally, make sure your environment has an install from the last version of those libraries.

To be able to share your model with the community and generate results like the one shown in the picture below via the inference API, there are a few more steps to follow.

First you have to store your authentication token from the Hugging Face website (sign up [here](https://huggingface.co/join) if you haven't already!) then uncomment the following cell and input your username and password (this only works on Colab, in a regular notebook, you need to do this in a terminal):
"""

# !huggingface-cli login

"""Then you need to install Git-LFS and setup Git if you haven't already. Uncomment the following instructions and adapt with your name and email:"""

!apt install git-lfs
!git config --global user.email "jirarotej@gmail.com"
!git config --global user.name "Jirarote Jirasirikul"

"""Make sure your version of Transformers is at least 4.8.1 since the functionality was introduced in that version:"""

import transformers

print(transformers.__version__)

from transformers import AutoTokenizer
from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer

"""You can find a script version of this notebook to fine-tune your model in a distributed fashion using multiple GPUs or TPUs [here](https://github.com/huggingface/transformers/tree/master/examples/text-classification).

# Fine-tuning a model on a text classification task

In this notebook, we will see how to fine-tune one of the [🤗 Transformers](https://github.com/huggingface/transformers) model to a text classification task of the [GLUE Benchmark](https://gluebenchmark.com/).
```

```
![Widget inference on a text classification
task](https://github.com/huggingface/notebooks/blob/master/examples/images/text_classification.png?raw=1)

The GLUE Benchmark is a group of nine classification tasks on sentences or pairs of sentences which are:

- [CoLA](https://nyu-mll.github.io/CoLA/) (Corpus of Linguistic Acceptability) Determine if a sentence is grammatically correct or not.is a  dataset containing sentences labeled grammatically correct or not.
- [MNLI](https://arxiv.org/abs/1704.05426) (Multi-Genre Natural Language Inference) Determine if a sentence entails, contradicts or is unrelated to a given hypothesis. (This dataset has two versions, one with the validation and test set coming from the same distribution, another called mismatched where the validation and test use out-of-domain data.)
- [MRPC](https://www.microsoft.com/en-us/download/details.aspx?id=52398) (Microsoft Research Paraphrase Corpus) Determine if two sentences are paraphrases from one another or not.
- [QNLI](https://rajpurkar.github.io/SQuAD-explorer/) (Question-answering Natural Language Inference) Determine if the answer to a question is in the second sentence or not. (This dataset is built from the SQuAD dataset.)
- [QQP](https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs) (Quora Question Pairs2) Determine if two questions are semantically equivalent or not.
- [RTE](https://aclweb.org/aclwiki/Recognizing_Textual_Entailment) (Recognizing Textual Entailment) Determine if a sentence entails a given hypothesis or not.
- [SST-2](https://nlp.stanford.edu/sentiment/index.html) (Stanford Sentiment Treebank) Determine if the sentence has a positive or negative sentiment.
- [STS-B](http://ixa2.si.ehu.es/stswiki/index.php/STSbenchmark) (Semantic Textual Similarity Benchmark) Determine the similarity of two sentences with a score from 1 to 5.
- [WNLI](https://cs.nyu.edu/faculty/davise/papers/WinogradSchemas/WS.html) (Winograd Natural Language Inference) Determine if a sentence with an anonymous pronoun and a sentence with this pronoun replaced are entailed or not. (This dataset is built from the Winograd Schema Challenge dataset.)

We will see how to easily load the dataset for each one of those tasks and use the `Trainer` API to fine-tune a model on it. Each task is named by its acronym, with `mnli-mm` standing for the mismatched version of MNLI (so same training set as `mnli` but different validation and test sets):
"""

# GLUE_TASKS = ["cola", "mnli", "mnli-mm", "mrpc", "qnli", "qqp", "rte", "sst2", "stsb", "wnli"]

"""This notebook is built to run on any of the tasks in the list above, with any model checkpoint from the [Model Hub](https://huggingface.co/models) as long as that model has a version with a classification head. Depending on you model and the GPU you are using, you might need to adjust the batch size to avoid out-of-memory errors. Set those three parameters, then the rest of the notebook should run smoothly:

## Loading the dataset

We will use the [🤗 Datasets](https://github.com/huggingface/datasets) library to download the data and get the metric we need to use for evaluation (to compare our model to the benchmark). This can be easily done with the functions `load_dataset` and `load_metric`.
"""

import json
import os
import pandas as pd
from datasets import load_dataset, load_metric, Dataset, DatasetDict, ClassLabel

"""Apart from `mnli-mm` being a special code, we can directly pass our task name to those functions. `load_dataset` will cache the dataset to avoid downloading it again the next time you run this cell."""

# actual_task = "mnli" if task == "mnli-mm" else task
# dataset = load_dataset("glue", actual_task)
# metric = load_metric('glue', actual_task)

"""### HoC"""

HOC_LABEL_LIST = ['label_IM', 'label_ID', 'label_CE', 'label_RI', 'label_GS', 'label_GI', 'label_A', 'label_CD', 'label_PS', 'label_TPI']

def extract_hoc_label(df_input):
    try:
        temp = df_input['labels'].str.split(',').apply(lambda x: [int(i.split('_')[1]) for i in x])
        temp_df = pd.DataFrame(temp.tolist())
        temp_df.columns = HOC_LABEL_LIST
        df_output = pd.concat([df_input,temp_df], axis=1)
        print("Extract HoC label")
        return df_output
    except:
        print("Something went wrong")

DATAPATH = "/content/drive/MyDrive/MinorThesis/"
DATASET = "HoC"
```

```
temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"train.tsv")
temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"dev.tsv")
temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test.tsv")

df_train = pd.read_csv(temppath_train, sep='\t')
df_test = pd.read_csv(temppath_test, sep='\t')
df_valid = pd.read_csv(temppath_valid, sep='\t')

df_train = extract_hoc_label(df_train)
df_test = extract_hoc_label(df_test)
df_valid = extract_hoc_label(df_valid)

dataset = DatasetDict({
    'train':Dataset.from_pandas(df_train),
    'validation':Dataset.from_pandas(df_valid),
    'test':Dataset.from_pandas(df_test),
})

dataset = dataset.remove_columns(['labels'])
# for col in HOC_LABEL_LIST:
#     dataset = dataset.class_encode_column(col)
# dataset.rename_column_("final_decision", "labels")
dataset.num_rows

# dataset = get_pubmedqa_fold(list_data_fold,df_test,0)
# dataset.num_rows

"""The `dataset` object itself is [`DatasetDict`](https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasetdict), which contains one key for the training, validation and test set (with more keys for the mismatched validation and test set in the special case of `mnli`).

To access an actual element, you need to select a split first, then give an index:
"""

dataset["train"].features

dataset["train"][0]

"""To get a sense of what the data looks like, the following function will show some examples picked randomly in the dataset."""

import datasets
import random
import pandas as pd
from IPython.display import display, HTML

def show_random_elements(dataset, num_examples=10):
    assert num_examples <= len(dataset), "Can't pick more elements than there are in the dataset."
    picks = []
    for _ in range(num_examples):
        pick = random.randint(0, len(dataset)-1)
        while pick in picks:
            pick = random.randint(0, len(dataset)-1)
        picks.append(pick)

    df = pd.DataFrame(dataset[picks])
    for column, typ in dataset.features.items():
        if isinstance(typ, datasets.ClassLabel):
            df[column] = df[column].transform(lambda i: typ.names[i])
    display(HTML(df.to_html()))

show_random_elements(dataset["train"])

"""The metric is an instance of
[`datasets.Metric`](https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasets.Metric):"""

# https://github.com/huggingface/datasets/tree/master/metrics
from datasets import load_metric
metric = load_metric("accuracy")

# metric = load_metric("https://github.com/huggingface/datasets/blob/master/metrics/accuracy/accuracy.py")
# Example of typical usage
# for batch in dataset:
#     inputs, references = batch
#     predictions = model(inputs)
```

```
#     metric.add_batch(predictions=predictions, references=references)
# score = metric.compute()

metric

"""You can call its `compute` method with your predictions and labels directly and it will return a dictionary with the metric(s) value:"""

import numpy as np

fake_preds = np.random.randint(0, 2, size=(64,))
fake_labels = np.random.randint(0, 2, size=(64,))
metric.compute(predictions=fake_preds, references=fake_labels)

"""## Fine-tuning the model"""

# # MY GLOBAL FUNCTION -

ENABLE_LOGS = 1
def print_log(*arg, log_type="Info"):
    global ENABLE_LOGS
    if(ENABLE_LOGS==1 or log_type!="Info"):
        print("["+log_type+"]"," ".join(str(x) for x in arg))

from sklearn.metrics import confusion_matrix, classification_report
class my_evaluator:
    LABELS_HOC_FULL = ['activating invasion and metastasis', 'avoiding immune destruction',
                'cellular energetics', 'enabling replicative immortality', 'evading growth suppressors',
                'genomic instability and mutation', 'inducing angiogenesis', 'resisting cell death',
                'sustaining proliferative signaling', 'tumor promoting inflammation']
    LABELS_HOC_SHORT = ['label_IM', 'label_ID',
                    'label_CE', 'label_RI', 'label_GS',
                    'label_GI', 'label_A', 'label_CD',
                    'label_PS', 'label_TPI']
    @classmethod
    def divide(self, x, y):
        return np.true_divide(x, y, out=np.zeros_like(x, dtype=np.float), where=y != 0)

    @classmethod
    def get_p_r_f_arrary(self, test_predict_label, test_true_label):
        num, cat = test_predict_label.shape
        # print(num,cat)
        acc_list = []
        prc_list = []
        rec_list = []
        f_score_list = []
        for i in range(num):
            # print(test_predict_label[i])
            # print(test_true_label[i])

            acc = accuracy_score(test_true_label[i], test_predict_label[i])
            prc,rec,f_score,_ = precision_recall_fscore_support(test_true_label[i], test_predict_label[i], average='macro')

            if prc == 0 and rec == 0:
                f_score = 0
            else:
                f_score = 2 * prc * rec / (prc + rec)

            acc_list.append(acc)
            prc_list.append(prc)
            rec_list.append(rec)
            f_score_list.append(f_score)

        # print(prc_list)
        # print(rec_list)

        mean_prc = np.mean(prc_list)
        mean_rec = np.mean(rec_list)
        f_score = self.divide(2 * mean_prc * mean_rec, (mean_prc + mean_rec))
        return mean_prc, mean_rec, f_score

    @classmethod
    def hoc_sentence2doc(self, input_df):
        # Output variables
        data = {}
```

```
    input_labels_count = dict(zip(self.LABELS_HOC_FULL, [0]*len(self.LABELS_HOC_FULL))) # sentence

    # Group sentence back into documents
    for i in range(len(input_df)):
        input_row = input_df.iloc[i]

        key = input_row['index'][:input_row['index'].find('_')]

        if key not in data:
            data[key] = set()

        if not pd.isna(input_row['labels']):
            for l in input_row['labels'].split(','):
                ind,val = l.split('_')
                if(val != '0'):
                    data[key].add(self.LABELS_HOC_FULL[int(ind)])
                    input_labels_count[self.LABELS_HOC_FULL[int(ind)]] += 1

    return data, input_labels_count

@classmethod
def hoc_labels2np(self, data):
    labels_list = dict(zip(self.LABELS_HOC_FULL, [[],[],[],[],[],[],[],[],[],[]]))

    y_np = []

    for k, v in data.items():
        # print(k)
        # print(true,pred)
        t = [0] * len(self.LABELS_HOC_FULL)
        for i in v:
            t[self.LABELS_HOC_FULL.index(i)] = 1

        y_np.append(t)

        for lab in self.LABELS_HOC_FULL:
            if(lab in v):
                labels_list[lab].append(1)
            else:
                labels_list[lab].append(0)

    return np.array(y_np),labels_list

@classmethod
def analysis_hoc(self, input_df):
    # Reformat to Paper evaluator format
    input_df = input_df[['filename_line','label']].copy()
    input_df.columns = ['index','labels']

    data, labels_counts_sen = self.hoc_sentence2doc(input_df)

    print_log('HoC Dataset Details')
    print_log('No. of Documents:',len(data))
    print_log('No. of Sentences:',len(input_df))
    # print(labels_counts_sen)

    y_np, labels_counts_doc = self.hoc_labels2np(data)
    # print(y_np)
    # print(labels_counts_doc)
    res = pd.DataFrame()
    for lab in self.LABELS_HOC_FULL:
        print_log(lab,sum(labels_counts_doc[lab]),len(labels_counts_doc[lab]))
        temp_df = pd.DataFrame([[sum(labels_counts_doc[lab]),len(labels_counts_doc[lab])]], columns=['sum','len'], index=[lab])
        res = res.append(temp_df)

    return res

@classmethod
def eval_hoc(self, input_df):
    print_log("eval hoc",log_type="Function")

    # Reformat to Paper evaluator format
    ## Label need to be in a format of list of 10 cancers in fixed order
    true_df = input_df[['filename_line','label']].copy()
```

```
        pred_df = input_df[['filename_line','prediction']].copy()
        true_df.columns = ['index','labels']
        pred_df.columns = ['index','labels']

        # Group sentence back into documents
        data_true, true_labels_count_sen = self.hoc_sentence2doc(true_df)
        data_pred, pred_labels_count_sen = self.hoc_sentence2doc(pred_df)

        # merge data_true/pred into format of {'key':(set(true),set(pred))}
        assert data_true.keys() == data_pred.keys(), 'Key mismatch'
        all_keys = set(data_true.keys()).union(data_pred.keys())

        data = {}
        for k in all_keys:
            data[k] = (data_true[k],data_pred[k])
        # print(data)
        assert len(data) == 371, 'There are 371 documents in the test set: %d' % len(data)

        print_log('HoC Dataset Details')
        print_log('No. of Documents:',len(data))
        print_log('No. of Sentences:',len(true_df),'/',len(pred_df))
        print(true_labels_count_sen,pred_labels_count_sen)

        # Write into dataframe
        res_count_sen = pd.DataFrame()
        for lab in self.LABELS_HOC_FULL:
            temp_df = pd.DataFrame([[true_labels_count_sen[lab],pred_labels_count_sen[lab],len(true_df)]],
columns=['sentence_count_label','sentence_count_pred','sentence_count_total'], index=[lab])
            res_count_sen = res_count_sen.append(temp_df)
        # print(res_count)

        y_test, true_labels_count_doc = self.hoc_labels2np(data_true)
        y_pred, pred_labels_count_doc = self.hoc_labels2np(data_pred)

        res_count_doc = pd.DataFrame()
        for lab in self.LABELS_HOC_FULL:
            temp_df =
pd.DataFrame([[sum(true_labels_count_doc[lab]),sum(pred_labels_count_doc[lab]),len(true_labels_count_doc[lab])]],
columns=['doc_count_label','doc_count_pred','doc_count_total'], index=[lab])
            res_count_doc = res_count_doc.append(temp_df)
        res_confmat = pd.DataFrame(columns=['tn','fp','fn','tp'])

        # print(true_labels_list,pred_labels_list)
        for lab in self.LABELS_HOC_FULL:
            # print(lab)
            df2 = pd.DataFrame([list(confusion_matrix(true_labels_count_doc[lab],pred_labels_count_doc[lab]).ravel())],
columns=['tn','fp','fn','tp'], index=[lab])
            res_confmat = res_confmat.append(df2)
        # print(res_confmat)
        df_res = pd.concat([res_confmat,res_count_sen,res_count_doc], axis=1)
        print(df_res)

        r, p, f1 = self.get_p_r_f_arrary(y_pred, y_test)
        print('Precision: {:.6f}'.format(p))
        print('Recall   : {:.6f}'.format(r))
        print('F1       : {:.6f}'.format(f1))
        return float(r), float(p), float(f1) , df_res

    @classmethod
    def multi_acc(self, y_pred, y_test):
        y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
        _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)

        correct_pred = (y_pred_tags == y_test).float()
        acc = correct_pred.sum() / len(correct_pred)

        acc = torch.round(acc * 100)

        return acc

    @classmethod
    def eval_pubmedqa(self, input_df):
        class2idx = {
            'no':0,
```

```
            'maybe':1,
            'yes':2
        }
        idx2class = {v: k for k, v in class2idx.items()}
        input_df['label'].replace(idx2class, inplace=True)
        input_df['prediction'].replace(idx2class, inplace=True)

        confusion_matrix_df = pd.DataFrame(confusion_matrix(input_df.label.values, input_df.prediction.values))
        class_report = classification_report(input_df.label.values, input_df.prediction.values, digits=3,output_dict = True)
        print(class_report)
        sns.heatmap(confusion_matrix_df, annot=True)
        return class_report , confusion_matrix_df

    @classmethod
    def eval_bioasq(self, input_df):
        class2idx = {
            'no':0,
            'yes':1
        }
        idx2class = {v: k for k, v in class2idx.items()}
        input_df['label'].replace(idx2class, inplace=True)
        input_df['prediction'].replace(idx2class, inplace=True)

        confusion_matrix_df = pd.DataFrame(confusion_matrix(input_df.label.values, input_df.prediction.values))
        class_report = classification_report(input_df.label.values, input_df.prediction.values, digits = 3,output_dict = True)
        print(class_report)
        sns.heatmap(confusion_matrix_df, annot=True)


        acc = accuracy_score(input_df.label.values, input_df.prediction.values)
        prc,rec,f_score,_ = precision_recall_fscore_support(input_df.label.values, input_df.prediction.values, average='macro')

        print("acc",acc)
        print("prc",prc)
        print("rec",rec)
        print("f_score",f_score)

        return class_report , confusion_matrix_df

# eval_hoc(temp_df)

"""#### For all folds"""

import torch
torch.cuda.empty_cache()

task = "HoC"
model_checkpoint = "microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext"
# model_checkpoint = "dmis-lab/biobert-base-cased-v1.1"
# model_checkpoint = "bert-base-uncased"

LEARNING_RATE = 1e-5
BATCH_SIZE = 8
NUM_LABELS = 2
MAX_LENGTH = 512

metric_name = "accuracy"
metric = load_metric("accuracy")

model_name = model_checkpoint.split("/")[-1]

def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    # predictions = predictions[:, 0]
    predictions = np.argmax(predictions, axis=1)
    return metric.compute(predictions=predictions, references=labels)

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)

def preprocess_function(examples):
    return tokenizer(examples["sentence"], truncation=True,padding=True,max_length=MAX_LENGTH)

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, matthews_corrcoef
from sklearn.metrics import precision_recall_fscore_support,accuracy_score
```

```
# r, p, f1, _ = my_evaluator.eval_hoc(output_df)

def format_output(list_output,filename="result_adapter_hoc.csv"):
    list_df_labels = []
    list_df_preds = []
    for k,v in list_output.items():
        print(k,v)
        y_pred = np.argmax(v.predictions, axis=1)
        temp_df = pd.DataFrame(data=y_pred, columns=[k])
        list_df_preds.append(temp_df)

        y_true = v.label_ids
        temp_df = pd.DataFrame(data=y_true, columns=[k])
        list_df_labels.append(temp_df)

    temp_df_lab = pd.DataFrame(data=dataset["test"]['index'], columns=["filename_line"])
    temp_df_lab


    list_cancer = ['label_IM', 'label_ID', 'label_CE', 'label_RI', 'label_GS', 'label_GI', 'label_A', 'label_CD', 'label_PS', 'label_TPI']

    temp_df = pd.concat(list_df_labels,axis=1)
    for col in temp_df.columns:
        print(col)
        temp_df[col] = str(list_cancer.index(col))+'_'+temp_df[col].astype(str)
    temp_df['label'] = temp_df.apply(lambda x: ','.join(x.dropna().values.tolist()), axis=1)
    temp_df

    temp_df2 = pd.concat(list_df_preds,axis=1)
    for col in temp_df2.columns:
        print(col)
        temp_df2[col] = str(list_cancer.index(col))+'_'+temp_df2[col].astype(str)
    temp_df2['prediction'] = temp_df2.apply(lambda x: ','.join(x.dropna().values.tolist()), axis=1)
    temp_df2

    temp_df3 = pd.concat([temp_df_lab,temp_df.label,temp_df2.prediction], axis=1)
    temp_df3

    temp_df3.to_csv(DATAPATH+filename)

    return temp_df3

# output_df = format_output(list_output,"result_adapter_hoc_"+model_name+"_"+LEARNING_RATE+".csv")

def train_adapter_hoc(LANG_MODEL,LEARNING_RATE):
    model_name = LANG_MODEL.split("/")[-1]
    print("--------",model_name,LEARNING_RATE,"--------")
    training_args = TrainingArguments(
        learning_rate=LEARNING_RATE,
        num_train_epochs=6,
        per_device_train_batch_size=BATCH_SIZE,
        per_device_eval_batch_size=BATCH_SIZE,
        eval_accumulation_steps=1,
        logging_steps=200,
        output_dir="./training_output",
        overwrite_output_dir=True,
        # The next line is important to ensure the dataset labels are properly passed to the model
        remove_unused_columns=False,
    )
    list_f1 = []
    list_output = {}
    for lab in HOC_LABEL_LIST:
        torch.cuda.empty_cache()

        print("###### LAB",lab,"######")

        HOC_LABEL_LIST_temp = HOC_LABEL_LIST.copy()
        HOC_LABEL_LIST_temp.remove(lab)

        # dataset_lab = dataset.remove_columns(HOC_LABEL_LIST[1:])

        dataset_lab = dataset.remove_columns(HOC_LABEL_LIST_temp)
        print("num_rows",dataset_lab.num_rows)
```

```
        # print(dataset_lab)

        encoded_dataset = dataset_lab.map(preprocess_function, batched=True)
        encoded_dataset.rename_column_(lab, "labels")
        encoded_dataset.set_format(type="torch", columns=["input_ids", "attention_mask","labels"])
        print(encoded_dataset)

        # dataset_lab.rename_column_(lab, "labels")
        # dataset_lab.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])
        # dataset_lab = dataset_lab.class_encode_column("labels")
        # encoded_dataset = dataset_lab.map(preprocess_function, batched=True)
        # encoded_dataset.set_format(type="torch", columns=["input_ids", "attention_mask","labels"])

        print("Setup model",model_checkpoint)
        config = AutoConfig.from_pretrained(
            model_checkpoint,
            num_labels=2,
        )
        model = AutoModelWithHeads.from_pretrained(
            model_checkpoint,
            config=config,
        )

        # Add a new adapter
        model.add_adapter("hoc")
        # Add a matching classification head
        model.add_classification_head(
            "hoc",
            num_labels=2,
            # id2label={ 0: "👎", 1: "👍"}
          )
        # Activate the adapter
        model.train_adapter("hoc")

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

        trainer.train()

        eval_pred = trainer.predict(encoded_dataset['test'])
        list_output[lab] = eval_pred

    output_df = format_output(list_output,"result_adapter_hoc_"+model_name+"_"+str(LEARNING_RATE)+".csv")
    r, p, f1, _ = my_evaluator.eval_hoc(output_df)
    print("")
    print("+++++","+++++")
    print("+++++","r",r,"+++++")
    print("+++++","p",p,"+++++")
    print("+++++","F1",f1,"+++++")
    print("+++++","+++++")
    print("")


for model in ["bert-base-uncased"]:
    for lr in [5e-5]:
        train_adapter_hoc(model,lr)

list_f1 = []
list_output = {}
for lab in HOC_LABEL_LIST:
    torch.cuda.empty_cache()
    print("###### LAB",lab,"######")

    HOC_LABEL_LIST_temp = HOC_LABEL_LIST.copy()
    HOC_LABEL_LIST_temp.remove(lab)

    dataset_lab = dataset.remove_columns(HOC_LABEL_LIST_temp)
    dataset_lab.rename_column_(lab, "labels")
    dataset_lab = dataset_lab.class_encode_column("labels")
```

```
    print("num_rows",dataset_lab.num_rows)
    print(dataset_lab['train'].features)

    encoded_dataset = dataset_lab.map(preprocess_function, batched=True)
    encoded_dataset.set_format(type="torch", columns=["input_ids", "attention_mask","labels"])

    model = AutoModelForSequenceClassification.from_pretrained(model_checkpoint, num_labels=NUM_LABELS)

    args = TrainingArguments(
        f"{model_name}-finetuned-{task}",
        evaluation_strategy = "epoch",
        save_strategy = "epoch",
        learning_rate=LEARNING_RATE,
        per_device_train_batch_size=BATCH_SIZE,
        per_device_eval_batch_size=BATCH_SIZE,
        eval_accumulation_steps=1,
        num_train_epochs=6,
        weight_decay=0.01,
        load_best_model_at_end=True,
        metric_for_best_model=metric_name,
        # The next line is important to ensure the dataset labels are properly passed to the model
        remove_unused_columns=True,
        # push_to_hub=True,
        # push_to_hub_model_id=f"{model_name}-finetuned-{task}",
    )

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

    trainer.train()

    commit_msg = ""
    commit_msg += "BATCH_SIZE="+str(BATCH_SIZE)+"\n"
    commit_msg += "LEARNING_RATE="+str(LEARNING_RATE)+"\n"
    commit_msg += "MAX_LENGTH="+str(MAX_LENGTH)+"\n"
    commit_msg += "LABEL="+lab+"\n"

    # trainer.push_to_hub(commit_msg)
    # output = trainer.evaluate(encoded_dataset['test'])
    # list_f1.append(output)

    eval_pred = trainer.predict(encoded_dataset['test'])
    list_output[lab] = eval_pred
    # temp = {}
    # temp['predictions'] = eval_pred['predictions']
    # temp['label_ids'] = eval_pred['label_ids']
    # list_output[lab] = temp

list_df_labels = []
list_df_preds = []
for k,v in list_output.items():
    print(k,v)
    y_pred = np.argmax(v.predictions, axis=1)
    temp_df = pd.DataFrame(data=y_pred, columns=[k])
    list_df_preds.append(temp_df)

    y_true = v.label_ids
    temp_df = pd.DataFrame(data=y_true, columns=[k])
    list_df_labels.append(temp_df)

temp_df_lab = pd.DataFrame(data=dataset["test"]['index'], columns=["filename_line"])
temp_df_lab

list_cancer = ['label_IM', 'label_ID', 'label_CE', 'label_RI', 'label_GS', 'label_GI', 'label_A', 'label_CD', 'label_PS', 'label_TPI']

temp_df = pd.concat(list_df_labels,axis=1)
for col in temp_df.columns:
```

```
    print(col)
    temp_df[col] = str(list_cancer.index(col))+'_'+temp_df[col].astype(str)
temp_df['label'] = temp_df.apply(lambda x: ','.join(x.dropna().values.tolist()), axis=1)
temp_df

temp_df2 = pd.concat(list_df_preds,axis=1)
for col in temp_df2.columns:
    print(col)
    temp_df2[col] = str(list_cancer.index(col))+'_'+temp_df2[col].astype(str)
temp_df2['prediction'] = temp_df2.apply(lambda x: ','.join(x.dropna().values.tolist()), axis=1)
temp_df2

temp_df3 = pd.concat([temp_df_lab,temp_df.label,temp_df2.prediction], axis=1)
temp_df3

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, matthews_corrcoef
from sklearn.metrics import precision_recall_fscore_support,accuracy_score

r, p, f1, _ = my_evaluator.eval_hoc(temp_df3)
```

4. text_classification_on_blurb_hoc_adapter.py

```
# -*- coding: utf-8 -*-
"""Text Classification on BLURB - HoC - Adapter

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1IawBTCxl-Yv_kdOPuvTSHNtK9fTjV907
"""

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)

from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')

"""If you're opening this Notebook on colab, you will probably need to install 🤗 Transformers and 🤗 Datasets. Uncomment the following cell and run it."""

!pip install -U adapter-transformers
!pip install datasets
# ! pip install datasets transformers

"""If you're opening this notebook locally, make sure your environment has an install from the last version of those libraries.

To be able to share your model with the community and generate results like the one shown in the picture below via the inference API, there are a few more steps to follow.

First you have to store your authentication token from the Hugging Face website (sign up [here](https://huggingface.co/join) if you haven't already!) then uncomment the following cell and input your username and password (this only works on Colab, in a regular notebook, you need to do this in a terminal):
"""

# !huggingface-cli login

"""Then you need to install Git-LFS and setup Git if you haven't already. Uncomment the following instructions and adapt with your name and email:"""

!apt install git-lfs
!git config --global user.email "jirarotej@gmail.com"
!git config --global user.name "Jirarote Jirasirikul"

"""Make sure your version of Transformers is at least 4.8.1 since the functionality was introduced in that version:"""

import transformers

print(transformers.__version__)

from transformers import AutoTokenizer
from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer, AutoModelWithHeads

"""You can find a script version of this notebook to fine-tune your model in a distributed fashion using multiple GPUs or TPUs [here](https://github.com/huggingface/transformers/tree/master/examples/text-classification).

# Fine-tuning a model on a text classification task

In this notebook, we will see how to fine-tune one of the [🤗 Transformers](https://github.com/huggingface/transformers) model to a
```

text classification task of the [GLUE Benchmark](https://gluebenchmark.com/).

The GLUE Benchmark is a group of nine classification tasks on sentences or pairs of sentences which are:

- [CoLA](https://nyu-mll.github.io/CoLA/) (Corpus of Linguistic Acceptability) Determine if a sentence is grammatically correct or not.is a dataset containing sentences labeled grammatically correct or not.
- [MNLI](https://arxiv.org/abs/1704.05426) (Multi-Genre Natural Language Inference) Determine if a sentence entails, contradicts or is unrelated to a given hypothesis. (This dataset has two versions, one with the validation and test set coming from the same distribution, another called mismatched where the validation and test use out-of-domain data.)
- [MRPC](https://www.microsoft.com/en-us/download/details.aspx?id=52398) (Microsoft Research Paraphrase Corpus) Determine if two sentences are paraphrases from one another or not.
- [QNLI](https://rajpurkar.github.io/SQuAD-explorer/) (Question-answering Natural Language Inference) Determine if the answer to a question is in the second sentence or not. (This dataset is built from the SQuAD dataset.)
- [QQP](https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs) (Quora Question Pairs2) Determine if two questions are semantically equivalent or not.
- [RTE](https://aclweb.org/aclwiki/Recognizing_Textual_Entailment) (Recognizing Textual Entailment) Determine if a sentence entails a given hypothesis or not.
- [SST-2](https://nlp.stanford.edu/sentiment/index.html) (Stanford Sentiment Treebank) Determine if the sentence has a positive or negative sentiment.
- [STS-B](http://ixa2.si.ehu.es/stswiki/index.php/STSbenchmark) (Semantic Textual Similarity Benchmark) Determine the similarity of two sentences with a score from 1 to 5.
- [WNLI](https://cs.nyu.edu/faculty/davise/papers/WinogradSchemas/WS.html) (Winograd Natural Language Inference) Determine if a sentence with an anonymous pronoun and a sentence with this pronoun replaced are entailed or not. (This dataset is built from the Winograd Schema Challenge dataset.)

We will see how to easily load the dataset for each one of those tasks and use the `Trainer` API to fine-tune a model on it. Each task is named by its acronym, with `mnli-mm` standing for the mismatched version of MNLI (so same training set as `mnli` but different validation and test sets):
"""

GLUE_TASKS = ["cola", "mnli", "mnli-mm", "mrpc", "qnli", "qqp", "rte", "sst2", "stsb", "wnli"]

"""This notebook is built to run on any of the tasks in the list above, with any model checkpoint from the [Model Hub](https://huggingface.co/models) as long as that model has a version with a classification head. Depending on you model and the GPU you are using, you might need to adjust the batch size to avoid out-of-memory errors. Set those three parameters, then the rest of the notebook should run smoothly:

Loading the dataset

We will use the [🤗 Datasets](https://github.com/huggingface/datasets) library to download the data and get the metric we need to use for evaluation (to compare our model to the benchmark). This can be easily done with the functions `load_dataset` and `load_metric`.
"""

import json
import os
import pandas as pd
from datasets import load_dataset, load_metric, Dataset, DatasetDict, ClassLabel

"""Apart from `mnli-mm` being a special code, we can directly pass our task name to those functions. `load_dataset` will cache the dataset to avoid downloading it again the next time you run this cell."""

actual_task = "mnli" if task == "mnli-mm" else task
dataset = load_dataset("glue", actual_task)
metric = load_metric('glue', actual_task)

"""### HoC"""

HOC_LABEL_LIST = ['label_IM', 'label_ID', 'label_CE', 'label_RI', 'label_GS', 'label_GI', 'label_A', 'label_CD', 'label_PS', 'label_TPI']

def extract_hoc_label(df_input):
    try:
        temp = df_input['labels'].str.split(',').apply(lambda x: [int(i.split('_')[1]) for i in x])
        temp_df = pd.DataFrame(temp.tolist())
        temp_df.columns = HOC_LABEL_LIST
        df_output = pd.concat([df_input,temp_df], axis=1)
        print("Extract HoC label")
        return df_output
    except:
        print("Something went wrong")

DATAPATH = "/content/drive/MyDrive/MinorThesis/"

```
DATASET = "HoC"

temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"train.tsv")
temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"dev.tsv")
temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test.tsv")

df_train = pd.read_csv(temppath_train, sep='\t')
df_test = pd.read_csv(temppath_test, sep='\t')
df_valid = pd.read_csv(temppath_valid, sep='\t')

df_train = extract_hoc_label(df_train)
df_test = extract_hoc_label(df_test)
df_valid = extract_hoc_label(df_valid)

dataset = DatasetDict({
    'train':Dataset.from_pandas(df_train),
    'validation':Dataset.from_pandas(df_valid),
    'test':Dataset.from_pandas(df_test),
})

dataset = dataset.remove_columns(['labels'])
# for col in HOC_LABEL_LIST:
#     dataset = dataset.class_encode_column(col)
# dataset.rename_column_("final_decision", "labels")
dataset.num_rows

# dataset = get_pubmedqa_fold(list_data_fold,df_test,0)
# dataset.num_rows

"""The `dataset` object itself is [`DatasetDict`](https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasetdict), which contains one key for the training, validation and test set (with more keys for the mismatched validation and test set in the special case of `mnli`).

To access an actual element, you need to select a split first, then give an index:
"""

dataset["train"].features

dataset

"""To get a sense of what the data looks like, the following function will show some examples picked randomly in the dataset."""

# import datasets
# import random
# import pandas as pd
# from IPython.display import display, HTML

# def show_random_elements(dataset, num_examples=10):
#     assert num_examples <= len(dataset), "Can't pick more elements than there are in the dataset."
#     picks = []
#     for _ in range(num_examples):
#         pick = random.randint(0, len(dataset)-1)
#         while pick in picks:
#             pick = random.randint(0, len(dataset)-1)
#         picks.append(pick)

#     df = pd.DataFrame(dataset[picks])
#     for column, typ in dataset.features.items():
#         if isinstance(typ, datasets.ClassLabel):
#             df[column] = df[column].transform(lambda i: typ.names[i])
#     display(HTML(df.to_html()))

# show_random_elements(dataset["train"])

"""The metric is an instance of
[`datasets.Metric`](https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasets.Metric):"""

# https://github.com/huggingface/datasets/tree/master/metrics
from datasets import load_metric
metric = load_metric("accuracy")

# metric = load_metric("https://github.com/huggingface/datasets/blob/master/metrics/accuracy/accuracy.py")
# Example of typical usage
# for batch in dataset:
```

```
#     inputs, references = batch
#     predictions = model(inputs)
#     metric.add_batch(predictions=predictions, references=references)
# score = metric.compute()

metric

"""You can call its `compute` method with your predictions and labels directly and it will return a dictionary with the metric(s) value:"""

import numpy as np

fake_preds = np.random.randint(0, 2, size=(64,))
fake_labels = np.random.randint(0, 2, size=(64,))
metric.compute(predictions=fake_preds, references=fake_labels)

"""## Fine-tuning the model"""

# # MY GLOBAL FUNCTION -

ENABLE_LOGS = 1
def print_log(*arg, log_type="Info"):
    global ENABLE_LOGS
    if(ENABLE_LOGS==1 or log_type!="Info"):
        print("["+log_type+"]"," ".join(str(x) for x in arg))

from sklearn.metrics import confusion_matrix, classification_report
class my_evaluator:
    LABELS_HOC_FULL = ['activating invasion and metastasis', 'avoiding immune destruction',
                'cellular energetics', 'enabling replicative immortality', 'evading growth suppressors',
                'genomic instability and mutation', 'inducing angiogenesis', 'resisting cell death',
                'sustaining proliferative signaling', 'tumor promoting inflammation']
    LABELS_HOC_SHORT = ['label_IM', 'label_ID',
                    'label_CE', 'label_RI', 'label_GS',
                    'label_GI', 'label_A', 'label_CD',
                    'label_PS', 'label_TPI']
    @classmethod
    def divide(self, x, y):
        return np.true_divide(x, y, out=np.zeros_like(x, dtype=np.float), where=y != 0)

    @classmethod
    def get_p_r_f_arrary(self, test_predict_label, test_true_label):
        num, cat = test_predict_label.shape
        # print(num,cat)
        acc_list = []
        prc_list = []
        rec_list = []
        f_score_list = []
        for i in range(num):
            # print(test_predict_label[i])
            # print(test_true_label[i])

            acc = accuracy_score(test_true_label[i], test_predict_label[i])
            prc,rec,f_score,_ = precision_recall_fscore_support(test_true_label[i], test_predict_label[i], average='macro')

            if prc == 0 and rec == 0:
                f_score = 0
            else:
                f_score = 2 * prc * rec / (prc + rec)

            acc_list.append(acc)
            prc_list.append(prc)
            rec_list.append(rec)
            f_score_list.append(f_score)

        # print(prc_list)
        # print(rec_list)

        mean_prc = np.mean(prc_list)
        mean_rec = np.mean(rec_list)
        f_score = self.divide(2 * mean_prc * mean_rec, (mean_prc + mean_rec))
        return mean_prc, mean_rec, f_score

    @classmethod
    def hoc_sentence2doc(self, input_df):
```

```
    # Output variables
    data = {}
    input_labels_count = dict(zip(self.LABELS_HOC_FULL, [0]*len(self.LABELS_HOC_FULL))) # sentence

    # Group sentence back into documents
    for i in range(len(input_df)):
        input_row = input_df.iloc[i]

        key = input_row['index'][:input_row['index'].find('_')]

        if key not in data:
            data[key] = set()

        if not pd.isna(input_row['labels']):
            for l in input_row['labels'].split(','):
                ind,val = l.split('_')
                if(val != '0'):
                    data[key].add(self.LABELS_HOC_FULL[int(ind)])
                    input_labels_count[self.LABELS_HOC_FULL[int(ind)]] += 1

    return data, input_labels_count

@classmethod
def hoc_labels2np(self, data):
    labels_list = dict(zip(self.LABELS_HOC_FULL, [[],[],[],[],[],[],[],[],[],[]]))

    y_np = []

    for k, v in data.items():
        # print(k)
        # print(true,pred)
        t = [0] * len(self.LABELS_HOC_FULL)
        for i in v:
            t[self.LABELS_HOC_FULL.index(i)] = 1

        y_np.append(t)

        for lab in self.LABELS_HOC_FULL:
            if(lab in v):
                labels_list[lab].append(1)
            else:
                labels_list[lab].append(0)

    return np.array(y_np),labels_list

@classmethod
def analysis_hoc(self, input_df):
    # Reformat to Paper evaluator format
    input_df = input_df[['filename_line','label']].copy()
    input_df.columns = ['index','labels']

    data, labels_counts_sen = self.hoc_sentence2doc(input_df)

    print_log('HoC Dataset Details')
    print_log('No. of Documents:',len(data))
    print_log('No. of Sentences:',len(input_df))
    # print(labels_counts_sen)

    y_np, labels_counts_doc = self.hoc_labels2np(data)
    # print(y_np)
    # print(labels_counts_doc)
    res = pd.DataFrame()
    for lab in self.LABELS_HOC_FULL:
        print_log(lab,sum(labels_counts_doc[lab]),len(labels_counts_doc[lab]))
        temp_df = pd.DataFrame([[sum(labels_counts_doc[lab]),len(labels_counts_doc[lab])]], columns=['sum','len'], index=[lab])
        res = res.append(temp_df)

    return res

@classmethod
def eval_hoc(self, input_df):
    print_log("eval hoc",log_type="Function")

    # Reformat to Paper evaluator format
```

```
        ## Label need to be in a format of list of 10 cancers in fixed order
        true_df = input_df[['filename_line','label']].copy()
        pred_df = input_df[['filename_line','prediction']].copy()
        true_df.columns = ['index','labels']
        pred_df.columns = ['index','labels']

        # Group sentence back into documents
        data_true, true_labels_count_sen = self.hoc_sentence2doc(true_df)
        data_pred, pred_labels_count_sen = self.hoc_sentence2doc(pred_df)

        # merge data_true/pred into format of {'key':(set(true),set(pred))}
        assert data_true.keys() == data_pred.keys(), 'Key mismatch'
        all_keys = set(data_true.keys()).union(data_pred.keys())

        data = {}
        for k in all_keys:
            data[k] = (data_true[k],data_pred[k])
        # print(data)
        assert len(data) == 371, 'There are 371 documents in the test set: %d' % len(data)

        print_log('HoC Dataset Details')
        print_log('No. of Documents:',len(data))
        print_log('No. of Sentences:',len(true_df),'/',len(pred_df))
        print(true_labels_count_sen,pred_labels_count_sen)

        # Write into dataframe
        res_count_sen = pd.DataFrame()
        for lab in self.LABELS_HOC_FULL:
            temp_df = pd.DataFrame([[true_labels_count_sen[lab],pred_labels_count_sen[lab],len(true_df)]],
columns=['sentence_count_label','sentence_count_pred','sentence_count_total'], index=[lab])
            res_count_sen = res_count_sen.append(temp_df)
        # print(res_count)

        y_test, true_labels_count_doc = self.hoc_labels2np(data_true)
        y_pred, pred_labels_count_doc = self.hoc_labels2np(data_pred)

        res_count_doc = pd.DataFrame()
        for lab in self.LABELS_HOC_FULL:
            temp_df =
pd.DataFrame([[sum(true_labels_count_doc[lab]),sum(pred_labels_count_doc[lab]),len(true_labels_count_doc[lab])]],
columns=['doc_count_label','doc_count_pred','doc_count_total'], index=[lab])
            res_count_doc = res_count_doc.append(temp_df)
        res_confmat = pd.DataFrame(columns=['tn','fp','fn','tp'])

        # print(true_labels_list,pred_labels_list)
        for lab in self.LABELS_HOC_FULL:
            # print(lab)
            df2 = pd.DataFrame([list(confusion_matrix(true_labels_count_doc[lab],pred_labels_count_doc[lab]).ravel())],
columns=['tn','fp','fn','tp'], index=[lab])
            res_confmat = res_confmat.append(df2)
        # print(res_confmat)
        df_res = pd.concat([res_confmat,res_count_sen,res_count_doc], axis=1)
        print(df_res)

        r, p, f1 = self.get_p_r_f_arrary(y_pred, y_test)
        print('Precision: {:.6f}'.format(p))
        print('Recall   : {:.6f}'.format(r))
        print('F1       : {:.6f}'.format(f1))
        return float(r), float(p), float(f1) , df_res

    @classmethod
    def multi_acc(self, y_pred, y_test):
        y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
        _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)

        correct_pred = (y_pred_tags == y_test).float()
        acc = correct_pred.sum() / len(correct_pred)

        acc = torch.round(acc * 100)

        return acc

    @classmethod
    def eval_pubmedqa(self, input_df):
```

```
        class2idx = {
            'no':0,
            'maybe':1,
            'yes':2
        }
        idx2class = {v: k for k, v in class2idx.items()}
        input_df['label'].replace(idx2class, inplace=True)
        input_df['prediction'].replace(idx2class, inplace=True)

        confusion_matrix_df = pd.DataFrame(confusion_matrix(input_df.label.values, input_df.prediction.values))
        class_report = classification_report(input_df.label.values, input_df.prediction.values, digits=3,output_dict = True)
        print(class_report)
        sns.heatmap(confusion_matrix_df, annot=True)
        return class_report , confusion_matrix_df

    @classmethod
    def eval_bioasq(self, input_df):
        class2idx = {
            'no':0,
            'yes':1
        }
        idx2class = {v: k for k, v in class2idx.items()}
        input_df['label'].replace(idx2class, inplace=True)
        input_df['prediction'].replace(idx2class, inplace=True)

        confusion_matrix_df = pd.DataFrame(confusion_matrix(input_df.label.values, input_df.prediction.values))
        class_report = classification_report(input_df.label.values, input_df.prediction.values, digits = 3,output_dict = True)
        print(class_report)
        sns.heatmap(confusion_matrix_df, annot=True)


        acc = accuracy_score(input_df.label.values, input_df.prediction.values)
        prc,rec,f_score,_ = precision_recall_fscore_support(input_df.label.values, input_df.prediction.values, average='macro')

        print("acc",acc)
        print("prc",prc)
        print("rec",rec)
        print("f_score",f_score)

        return class_report , confusion_matrix_df

# eval_hoc(temp_df)

"""#### For all folds"""

import torch
torch.cuda.empty_cache()

task = "HoC"
# model_checkpoint = "microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext"
# model_checkpoint = "dmis-lab/biobert-base-cased-v1.1"
model_checkpoint = "bert-base-uncased"

LEARNING_RATE = 1e-5
BATCH_SIZE = 8
NUM_LABELS = 2
MAX_LENGTH = 512

metric_name = "accuracy"
metric = load_metric("accuracy")

model_name = model_checkpoint.split("/")[-1]

from transformers import TrainingArguments, Trainer, EvalPrediction

def compute_accuracy(p: EvalPrediction):
    preds = np.argmax(p.predictions, axis=1)
    return {"acc": (preds == p.label_ids).mean()}

def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    # predictions = predictions[:, 0]
    predictions = np.argmax(predictions, axis=1)
    return metric.compute(predictions=predictions, references=labels)
```

```
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)

def preprocess_function(examples):
    # return tokenizer(examples["sentence"], truncation=True,padding=True,max_length=MAX_LENGTH)
    return tokenizer(examples["sentence"], max_length=MAX_LENGTH, truncation=True, padding="max_length")

from transformers import AutoModelWithHeads,AutoConfig

training_args = TrainingArguments(
    learning_rate=LEARNING_RATE,
    num_train_epochs=6,
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    eval_accumulation_steps=1,
    logging_steps=200,
    output_dir="./training_output",
    overwrite_output_dir=True,
    # The next line is important to ensure the dataset labels are properly passed to the model
    remove_unused_columns=False,
)

list_f1 = []
list_output = {}
for lab in HOC_LABEL_LIST:
    torch.cuda.empty_cache()

    print("###### LAB",lab,"######")

    HOC_LABEL_LIST_temp = HOC_LABEL_LIST.copy()
    HOC_LABEL_LIST_temp.remove(lab)

    # dataset_lab = dataset.remove_columns(HOC_LABEL_LIST[1:])

    dataset_lab = dataset.remove_columns(HOC_LABEL_LIST_temp)
    print("num_rows",dataset_lab.num_rows)
    # print(dataset_lab)

    encoded_dataset = dataset_lab.map(preprocess_function, batched=True)
    encoded_dataset.rename_column_(lab, "labels")
    encoded_dataset.set_format(type="torch", columns=["input_ids", "attention_mask","labels"])
    print(encoded_dataset)

    # dataset_lab.rename_column_(lab, "labels")
    # dataset_lab.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])
    # dataset_lab = dataset_lab.class_encode_column("labels")
    # encoded_dataset = dataset_lab.map(preprocess_function, batched=True)
    # encoded_dataset.set_format(type="torch", columns=["input_ids", "attention_mask","labels"])

    print("Setup model",model_checkpoint)
    config = AutoConfig.from_pretrained(
        model_checkpoint,
        num_labels=2,
    )
    model = AutoModelWithHeads.from_pretrained(
        model_checkpoint,
        config=config,
    )

    # Add a new adapter
    model.add_adapter("hoc")
    # Add a matching classification head
    model.add_classification_head(
        "hoc",
        num_labels=2,
        # id2label={ 0: "👎", 1: "👍"}
      )
    # Activate the adapter
    model.train_adapter("hoc")


    trainer = Trainer(
        model=model,
        args=training_args,
```

```
        train_dataset=encoded_dataset["train"],
        eval_dataset=encoded_dataset["validation"],
        compute_metrics=compute_accuracy,
    )

    trainer.train()

    # # model = AutoModelForSequenceClassification.from_pretrained(model_checkpoint, num_labels=NUM_LABELS)
    # model = AutoModelWithHeads.from_pretrained(model_checkpoint)
    # model.add_adapter("hoc")
    # model.train_adapter("hoc")
    # model.add_classification_head("hoc", num_labels=2)
    # model.set_active_adapters("hoc")
    # # model.add_adapter("HoC")
    # # model.train_adapter("HoC")

    # args = TrainingArguments(
    #         f"{model_name}-finetuned-{task}-{lab}",
    #         evaluation_strategy = "epoch",
    #         save_strategy = "epoch",
    #         learning_rate=LEARNING_RATE,
    #         per_device_train_batch_size=BATCH_SIZE,
    #         per_device_eval_batch_size=BATCH_SIZE,
    #         eval_accumulation_steps=1,
    #         num_train_epochs=1,
    #         weight_decay=0.01,
    #         load_best_model_at_end=True,
    #         metric_for_best_model=metric_name,
    #         # The next line is important to ensure the dataset labels are properly passed to the model
    #         remove_unused_columns=True,
    #         # push_to_hub=True,
    #         # push_to_hub_model_id=f"{model_name}-finetuned-{task}",
    #     )

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

    # trainer.train()

    # commit_msg = ""
    # commit_msg += "BATCH_SIZE="+str(BATCH_SIZE)+"\n"
    # commit_msg += "LEARNING_RATE="+str(LEARNING_RATE)+"\n"
    # commit_msg += "MAX_LENGTH="+str(MAX_LENGTH)+"\n"
    # commit_msg += "LABEL="+lab+"\n"

    # # trainer.push_to_hub(commit_msg)
    # # output = trainer.evaluate(encoded_dataset['test'])
    # # list_f1.append(output)

    eval_pred = trainer.predict(encoded_dataset['test'])
    list_output[lab] = eval_pred
    # # temp = {}
    # # temp['predictions'] = eval_pred['predictions']
    # # temp['label_ids'] = eval_pred['label_ids']
    # # list_output[lab] = temp

# y_pred = np.argmax(list_output['label_IM'].predictions, axis=1)
# y_true = list_output['label_IM'].label_ids

# a = np.bincount(y_true)
# b = np.bincount(y_pred)
# print("count value true (0/1)",a,"pred (0/1)",b)

def format_output(list_output,filename="result_adapter_hoc.csv"):
    list_df_labels = []
    list_df_preds = []
    for k,v in list_output.items():
        print(k,v)
```

```
        y_pred = np.argmax(v.predictions, axis=1)
        temp_df = pd.DataFrame(data=y_pred, columns=[k])
        list_df_preds.append(temp_df)

        y_true = v.label_ids
        temp_df = pd.DataFrame(data=y_true, columns=[k])
        list_df_labels.append(temp_df)

    temp_df_lab = pd.DataFrame(data=dataset["test"]['index'], columns=["filename_line"])
    temp_df_lab


    list_cancer = ['label_IM', 'label_ID', 'label_CE', 'label_RI', 'label_GS', 'label_GI', 'label_A', 'label_CD', 'label_PS', 'label_TPI']

    temp_df = pd.concat(list_df_labels,axis=1)
    for col in temp_df.columns:
        print(col)
        temp_df[col] = str(list_cancer.index(col))+'_'+temp_df[col].astype(str)
    temp_df['label'] = temp_df.apply(lambda x: ','.join(x.dropna().values.tolist()), axis=1)
    temp_df

    temp_df2 = pd.concat(list_df_preds,axis=1)
    for col in temp_df2.columns:
        print(col)
        temp_df2[col] = str(list_cancer.index(col))+'_'+temp_df2[col].astype(str)
    temp_df2['prediction'] = temp_df2.apply(lambda x: ','.join(x.dropna().values.tolist()), axis=1)
    temp_df2

    temp_df3 = pd.concat([temp_df_lab,temp_df.label,temp_df2.prediction], axis=1)
    temp_df3

    temp_df3.to_csv(DATAPATH+filename)

    return temp_df3

output_df = format_output(list_output,"result_adapter_hoc_"+model_name+"_"+str(LEARNING_RATE)+".csv")



from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, matthews_corrcoef
from sklearn.metrics import precision_recall_fscore_support,accuracy_score

r, p, f1, _ = my_evaluator.eval_hoc(output_df)

# DATAPATH = "/content/drive/MyDrive/MinorThesis/"
# with open(os.path.join(DATAPATH,"data.json"), 'w') as fp:
#     json.dump(list_output, fp)

# eval_pred = trainer.predict(encoded_dataset['test'])

# eval_pred.predictions

# predictions = eval_pred.predictions
# # , labels = eval_pred
# # predictions = predictions[:, 0]
# predictions = np.argmax(predictions, axis=1)
# predictions

list_output

# np.mean(list_acc)
```

5. text_classification_on_blurb_pubmedqa_traditional_fine_tuned.py

```
# -*- coding: utf-8 -*-
"""Text Classification on BLURB - PubMedQA - Traditional Fine-tuned

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1kVgitxrG3QfCo25oMSko-xAwhfWJG-vU
"""

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# !kill process_id

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)

from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')

"""If you're opening this Notebook on colab, you will probably need to install 🤗 Transformers and 🤗 Datasets. Uncomment the following cell and run it."""

! pip install datasets transformers

"""If you're opening this notebook locally, make sure your environment has an install from the last version of those libraries.

To be able to share your model with the community and generate results like the one shown in the picture below via the inference API, there are a few more steps to follow.

First you have to store your authentication token from the Hugging Face website (sign up [here](https://huggingface.co/join) if you haven't already!) then uncomment the following cell and input your username and password (this only works on Colab, in a regular notebook, you need to do this in a terminal):
"""

# !huggingface-cli login

"""Then you need to install Git-LFS and setup Git if you haven't already. Uncomment the following instructions and adapt with your name and email:"""

!apt install git-lfs
!git config --global user.email "jirarotej@gmail.com"
!git config --global user.name "Jirarote Jirasirikul"

"""Make sure your version of Transformers is at least 4.8.1 since the functionality was introduced in that version:"""

import transformers

print(transformers.__version__)

from transformers import AutoTokenizer
from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer

"""You can find a script version of this notebook to fine-tune your model in a distributed fashion using multiple GPUs or TPUs [here](https://github.com/huggingface/transformers/tree/master/examples/text-classification).

# Fine-tuning a model on a text classification task

In this notebook, we will see how to fine-tune one of the [🤗 Transformers](https://github.com/huggingface/transformers) model to a
```

text classification task of the [GLUE Benchmark](https://gluebenchmark.com/).

The GLUE Benchmark is a group of nine classification tasks on sentences or pairs of sentences which are:

- [CoLA](https://nyu-mll.github.io/CoLA/) (Corpus of Linguistic Acceptability) Determine if a sentence is grammatically correct or not.is a dataset containing sentences labeled grammatically correct or not.
- [MNLI](https://arxiv.org/abs/1704.05426) (Multi-Genre Natural Language Inference) Determine if a sentence entails, contradicts or is unrelated to a given hypothesis. (This dataset has two versions, one with the validation and test set coming from the same distribution, another called mismatched where the validation and test use out-of-domain data.)
- [MRPC](https://www.microsoft.com/en-us/download/details.aspx?id=52398) (Microsoft Research Paraphrase Corpus) Determine if two sentences are paraphrases from one another or not.
- [QNLI](https://rajpurkar.github.io/SQuAD-explorer/) (Question-answering Natural Language Inference) Determine if the answer to a question is in the second sentence or not. (This dataset is built from the SQuAD dataset.)
- [QQP](https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs) (Quora Question Pairs2) Determine if two questions are semantically equivalent or not.
- [RTE](https://aclweb.org/aclwiki/Recognizing_Textual_Entailment) (Recognizing Textual Entailment) Determine if a sentence entails a given hypothesis or not.
- [SST-2](https://nlp.stanford.edu/sentiment/index.html) (Stanford Sentiment Treebank) Determine if the sentence has a positive or negative sentiment.
- [STS-B](http://ixa2.si.ehu.es/stswiki/index.php/STSbenchmark) (Semantic Textual Similarity Benchmark) Determine the similarity of two sentences with a score from 1 to 5.
- [WNLI](https://cs.nyu.edu/faculty/davise/papers/WinogradSchemas/WS.html) (Winograd Natural Language Inference) Determine if a sentence with an anonymous pronoun and a sentence with this pronoun replaced are entailed or not. (This dataset is built from the Winograd Schema Challenge dataset.)

We will see how to easily load the dataset for each one of those tasks and use the `Trainer` API to fine-tune a model on it. Each task is named by its acronym, with `mnli-mm` standing for the mismatched version of MNLI (so same training set as `mnli` but different validation and test sets):
"""

GLUE_TASKS = ["cola", "mnli", "mnli-mm", "mrpc", "qnli", "qqp", "rte", "sst2", "stsb", "wnli"]

"""This notebook is built to run on any of the tasks in the list above, with any model checkpoint from the [Model Hub](https://huggingface.co/models) as long as that model has a version with a classification head. Depending on you model and the GPU you are using, you might need to adjust the batch size to avoid out-of-memory errors. Set those three parameters, then the rest of the notebook should run smoothly:

Loading the dataset

We will use the [🤗 Datasets](https://github.com/huggingface/datasets) library to download the data and get the metric we need to use for evaluation (to compare our model to the benchmark). This can be easily done with the functions `load_dataset` and `load_metric`.
"""

import os
import pandas as pd
from datasets import load_dataset, load_metric, Dataset, DatasetDict, ClassLabel

"""Apart from `mnli-mm` being a special code, we can directly pass our task name to those functions. `load_dataset` will cache the dataset to avoid downloading it again the next time you run this cell."""

actual_task = "mnli" if task == "mnli-mm" else task
dataset = load_dataset("glue", actual_task)
metric = load_metric('glue', actual_task)

"""### pubmedqa"""

DATAPATH = "/content/drive/MyDrive/MinorThesis/"
DATASET = "pubmedqa"

# Train & Valid
list_data_fold = []
for i in range(10): # We merge dataset to generate trained (Separate later using index - filename_line)
    temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"train_set.json")
    temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"dev_set.json")

    df_temp_train = pd.read_json(temppath_train).transpose()
    df_temp_valid = pd.read_json(temppath_valid).transpose()
    list_data_fold.append((df_temp_train,df_temp_valid))
    # print(df_temp_train.shape,df_temp_valid.shape)

Test

```
temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,'test_set.json')
df_test = pd.read_json(temppath_test).transpose()

def get_pubmedqa_fold(list_data_fold,data_test,fold_no = 0):
    # temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test_set.json")
    df_train = list_data_fold[fold_no][0]
    df_valid = list_data_fold[fold_no][1]
    df_test = data_test

    df_train['CONTEXTS_JOINED'] = df_train.CONTEXTS.apply(lambda x : (' ').join(x))
    df_valid['CONTEXTS_JOINED'] = df_valid.CONTEXTS.apply(lambda x : (' ').join(x))
    df_test['CONTEXTS_JOINED'] = df_test.CONTEXTS.apply(lambda x : (' ').join(x))

    # # REASONING REQUIRED
    # df_train_mod =
df_train[['QUESTION','CONTEXTS','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
    # df_train_mod['text'] = df_train_mod.QUESTION +". "+ df_train_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question before
    # # df_train_mod['text'] = df_train_mod.CONTEXTS.apply(lambda x : (' ').join(x)) +" "+ df_train_mod.QUESTION  #question after
    # df_train_mod.drop(columns=['QUESTION','CONTEXTS'],inplace=True)
    # df_train_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']

    # df_valid_mod =
df_valid[['QUESTION','CONTEXTS','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
    # df_valid_mod['text'] = df_valid_mod.QUESTION +". "+ df_valid_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question before
    # # df_train_mod['text'] = df_valid_mod.CONTEXTS.apply(lambda x : (' ').join(x)) +" "+ df_valid_mod.QUESTION  #question after
    # df_valid_mod.drop(columns=['QUESTION','CONTEXTS'],inplace=True)
    # df_valid_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']

    # df_test_mod =
df_test[['QUESTION','CONTEXTS','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
    # df_test_mod['text'] = df_test_mod.QUESTION +". "+ df_test_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question before
    # # df_test_mod['text'] = df_test_mod.CONTEXTS.apply(lambda x : (' ').join(x)) +" "+ df_test_mod.QUESTION  #question after
    # df_test_mod.drop(columns=['QUESTION','CONTEXTS'],inplace=True)
    # df_test_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']

    # id2label={ 0: "no", 1: "maybe", 2: "yes"}
    # label2id = dict((v,k) for k,v in id2label.items())
    # df_train_mod['label'] = df_train_mod.label.apply(lambda x : label2id[x])
    # df_valid_mod['label'] = df_valid_mod.label.apply(lambda x : label2id[x])
    # df_test_mod['label'] = df_test_mod.label.apply(lambda x : label2id[x])

    dataset = DatasetDict({
        'train':Dataset.from_pandas(df_train),
        'validation':Dataset.from_pandas(df_valid),
        'test':Dataset.from_pandas(df_test),
    })

    dataset.remove_columns_(['CONTEXTS','LABELS', 'MESHES','YEAR', 'reasoning_required_pred', 'reasoning_free_pred',
'LONG_ANSWER','__index_level_0__'])
    dataset.rename_column_("final_decision", "labels")
    return dataset.class_encode_column("labels")

dataset = get_pubmedqa_fold(list_data_fold,df_test,0)
dataset.num_rows

"""The `dataset` object itself is [`DatasetDict`](https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasetdict),
which contains one key for the training, validation and test set (with more keys for the mismatched validation and test set in the special
case of `mnli`).

To access an actual element, you need to select a split first, then give an index:
"""

dataset["train"].features

dataset["train"][0]

"""To get a sense of what the data looks like, the following function will show some examples picked randomly in the dataset."""

import datasets
import random
import pandas as pd
from IPython.display import display, HTML

def show_random_elements(dataset, num_examples=10):
```

```
    assert num_examples <= len(dataset), "Can't pick more elements than there are in the dataset."
    picks = []
    for _ in range(num_examples):
        pick = random.randint(0, len(dataset)-1)
        while pick in picks:
            pick = random.randint(0, len(dataset)-1)
        picks.append(pick)

    df = pd.DataFrame(dataset[picks])
    for column, typ in dataset.features.items():
        if isinstance(typ, datasets.ClassLabel):
            df[column] = df[column].transform(lambda i: typ.names[i])
    display(HTML(df.to_html()))

# show_random_elements(dataset["train"])

"""The metric is an instance of
[`datasets.Metric`](https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasets.Metric):"""

# https://github.com/huggingface/datasets/tree/master/metrics
from datasets import load_metric
metric = load_metric("accuracy")

# metric = load_metric("https://github.com/huggingface/datasets/blob/master/metrics/accuracy/accuracy.py")
# Example of typical usage
# for batch in dataset:
#     inputs, references = batch
#     predictions = model(inputs)
#     metric.add_batch(predictions=predictions, references=references)
# score = metric.compute()

metric

"""You can call its `compute` method with your predictions and labels directly and it will return a dictionary with the metric(s) value:"""

import numpy as np

fake_preds = np.random.randint(0, 3, size=(64,))
fake_labels = np.random.randint(0, 3, size=(64,))
metric.compute(predictions=fake_preds, references=fake_labels)

"""## Fine-tuning the model"""

def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    # predictions = predictions[:, 0]
    predictions = np.argmax(predictions, axis=1)
    return metric.compute(predictions=predictions, references=labels)

"""#### For all folds"""

import torch
torch.cuda.empty_cache()

task = "pubmedqa"
model_checkpoint = "microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext"
# model_checkpoint = "dmis-lab/biobert-base-cased-v1.1"
# model_checkpoint = "dmis-lab/biobert-v1.1"

LEARNING_RATE = 3e-3
BATCH_SIZE = 8
NUM_LABELS = 3
MAX_LENGTH = 512

metric_name = "accuracy"
model_name = model_checkpoint.split("/")[-1]

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)

def preprocess_function(examples):
    return tokenizer(examples["QUESTION"], examples["CONTEXTS_JOINED"],
truncation=True,padding=True,max_length=MAX_LENGTH)

list_acc = []
```

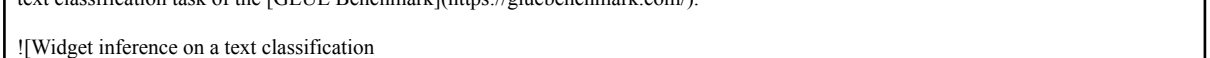
```
for i in range(10):
    torch.cuda.empty_cache()
    print("###### FOLD",i,"######")
    dataset = get_pubmedqa_fold(list_data_fold,df_test,i)
    print("num_rows",dataset.num_rows)

    encoded_dataset = dataset.map(preprocess_function, batched=True)
    encoded_dataset.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])

    model = AutoModelForSequenceClassification.from_pretrained(model_checkpoint, num_labels=NUM_LABELS)

    args = TrainingArguments(
        f"{model_name}-finetuned-{task}",
        evaluation_strategy = "epoch",
        save_strategy = "epoch",
        learning_rate=LEARNING_RATE,
        per_device_train_batch_size=BATCH_SIZE,
        per_device_eval_batch_size=BATCH_SIZE,
        eval_accumulation_steps=1,
        num_train_epochs=10,
        weight_decay=0.01,
        load_best_model_at_end=True,
        metric_for_best_model=metric_name,
        # The next line is important to ensure the dataset labels are properly passed to the model
        remove_unused_columns=True,
        # push_to_hub=True,
        # push_to_hub_model_id=f"{model_name}-finetuned-{task}-2",
    )

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

    trainer.train()

    commit_msg = ""
    commit_msg += "BATCH_SIZE="+str(BATCH_SIZE)+"\n"
    commit_msg += "LEARNING_RATE="+str(LEARNING_RATE)+"\n"
    commit_msg += "MAX_LENGTH="+str(MAX_LENGTH)+"\n"
    commit_msg += "FOLD="+str(i)+"\n"

    # trainer.push_to_hub(commit_msg)
    output = trainer.evaluate(encoded_dataset['test'])
    list_acc.append(output['eval_accuracy'])

list_acc

np.mean(list_acc)
```

6. text_classification_on_blurb_pubmedqa_adapter.py

```
# -*- coding: utf-8 -*-
"""Text Classification on BLURB - PubMedQA - Adapter

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1FpGNEl5VtrEmCFlcGBAVYeKwHgJ8OEnZ
"""

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)

from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')

"""If you're opening this Notebook on colab, you will probably need to install 🤗 Transformers and 🤗 Datasets. Uncomment the following cell and run it."""

# !pip install -U adapter-transformers
! pip install datasets

! pip install datasets transformers

"""If you're opening this notebook locally, make sure your environment has an install from the last version of those libraries.

To be able to share your model with the community and generate results like the one shown in the picture below via the inference API, there are a few more steps to follow.

First you have to store your authentication token from the Hugging Face website (sign up [here](https://huggingface.co/join) if you haven't already!) then uncomment the following cell and input your username and password (this only works on Colab, in a regular notebook, you need to do this in a terminal):
"""

# !huggingface-cli login

"""Then you need to install Git-LFS and setup Git if you haven't already. Uncomment the following instructions and adapt with your name and email:"""

!apt install git-lfs
!git config --global user.email "jirarotej@gmail.com"
!git config --global user.name "Jirarote Jirasirikul"

"""Make sure your version of Transformers is at least 4.8.1 since the functionality was introduced in that version:"""

import transformers

print(transformers.__version__)

from transformers import AutoTokenizer
from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer

# Adapter only
# from transformers import AutoModelWithHeads,AutoConfig

"""You can find a script version of this notebook to fine-tune your model in a distributed fashion using multiple GPUs or TPUs [here](https://github.com/huggingface/transformers/tree/master/examples/text-classification).
```

Fine-tuning a model on a text classification task

In this notebook, we will see how to fine-tune one of the [🤗 Transformers](https://github.com/huggingface/transformers) model to a text classification task of the [GLUE Benchmark](https://gluebenchmark.com/).



The GLUE Benchmark is a group of nine classification tasks on sentences or pairs of sentences which are:

- [CoLA](https://nyu-mll.github.io/CoLA/) (Corpus of Linguistic Acceptability) Determine if a sentence is grammatically correct or not.is a dataset containing sentences labeled grammatically correct or not.
- [MNLI](https://arxiv.org/abs/1704.05426) (Multi-Genre Natural Language Inference) Determine if a sentence entails, contradicts or is unrelated to a given hypothesis. (This dataset has two versions, one with the validation and test set coming from the same distribution, another called mismatched where the validation and test use out-of-domain data.)
- [MRPC](https://www.microsoft.com/en-us/download/details.aspx?id=52398) (Microsoft Research Paraphrase Corpus) Determine if two sentences are paraphrases from one another or not.
- [QNLI](https://rajpurkar.github.io/SQuAD-explorer/) (Question-answering Natural Language Inference) Determine if the answer to a question is in the second sentence or not. (This dataset is built from the SQuAD dataset.)
- [QQP](https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs) (Quora Question Pairs2) Determine if two questions are semantically equivalent or not.
- [RTE](https://aclweb.org/aclwiki/Recognizing_Textual_Entailment) (Recognizing Textual Entailment) Determine if a sentence entails a given hypothesis or not.
- [SST-2](https://nlp.stanford.edu/sentiment/index.html) (Stanford Sentiment Treebank) Determine if the sentence has a positive or negative sentiment.
- [STS-B](http://ixa2.si.ehu.es/stswiki/index.php/STSbenchmark) (Semantic Textual Similarity Benchmark) Determine the similarity of two sentences with a score from 1 to 5.
- [WNLI](https://cs.nyu.edu/faculty/davise/papers/WinogradSchemas/WS.html) (Winograd Natural Language Inference) Determine if a sentence with an anonymous pronoun and a sentence with this pronoun replaced are entailed or not. (This dataset is built from the Winograd Schema Challenge dataset.)

We will see how to easily load the dataset for each one of those tasks and use the `Trainer` API to fine-tune a model on it. Each task is named by its acronym, with `mnli-mm` standing for the mismatched version of MNLI (so same training set as `mnli` but different validation and test sets):
"""

GLUE_TASKS = ["cola", "mnli", "mnli-mm", "mrpc", "qnli", "qqp", "rte", "sst2", "stsb", "wnli"]

"""This notebook is built to run on any of the tasks in the list above, with any model checkpoint from the [Model Hub](https://huggingface.co/models) as long as that model has a version with a classification head. Depending on you model and the GPU you are using, you might need to adjust the batch size to avoid out-of-memory errors. Set those three parameters, then the rest of the notebook should run smoothly:

Loading the dataset

We will use the [🤗 Datasets](https://github.com/huggingface/datasets) library to download the data and get the metric we need to use for evaluation (to compare our model to the benchmark). This can be easily done with the functions `load_dataset` and `load_metric`.
"""

import os
import pandas as pd
from datasets import load_dataset, load_metric, Dataset, DatasetDict, ClassLabel

"""Apart from `mnli-mm` being a special code, we can directly pass our task name to those functions. `load_dataset` will cache the dataset to avoid downloading it again the next time you run this cell."""

actual_task = "mnli" if task == "mnli-mm" else task
dataset = load_dataset("glue", actual_task)
metric = load_metric('glue', actual_task)

"""### pubmedqa"""

DATAPATH = "/content/drive/MyDrive/MinorThesis/"
DATASET = "pubmedqa"

# Train & Valid
list_data_fold = []
for i in range(10): # We merge dataset to generate trained (Separate later using index - filename_line)
    temppath_train = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"train_set.json")
    temppath_valid = os.path.join(DATAPATH,"datasets","raw",DATASET,"pqal_fold"+str(i),"dev_set.json")

    df_temp_train = pd.read_json(temppath_train).transpose()
    df_temp_valid = pd.read_json(temppath_valid).transpose()

```
    list_data_fold.append((df_temp_train,df_temp_valid))
    # print(df_temp_train.shape,df_temp_valid.shape)

# Test
temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,'test_set.json')
df_test = pd.read_json(temppath_test).transpose()

def get_pubmedqa_fold(list_data_fold,data_test,fold_no = 0):
    # temppath_test = os.path.join(DATAPATH,"datasets","raw",DATASET,"test_set.json")
    df_train = list_data_fold[fold_no][0]
    df_valid = list_data_fold[fold_no][1]
    df_test = data_test

    df_train['CONTEXTS_JOINED'] = df_train.CONTEXTS.apply(lambda x : (' ').join(x))
    df_valid['CONTEXTS_JOINED'] = df_valid.CONTEXTS.apply(lambda x : (' ').join(x))
    df_test['CONTEXTS_JOINED'] = df_test.CONTEXTS.apply(lambda x : (' ').join(x))

    # # REASONING REQUIRED
    # df_train_mod =
df_train[['QUESTION','CONTEXTS','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
    # df_train_mod['text'] = df_train_mod.QUESTION +". "+ df_train_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question before
    # # df_train_mod['text'] = df_train_mod.CONTEXTS.apply(lambda x : (' ').join(x)) +" "+ df_train_mod.QUESTION  #question after
    # df_train_mod.drop(columns=['QUESTION','CONTEXTS'],inplace=True)
    # df_train_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']

    # df_valid_mod =
df_valid[['QUESTION','CONTEXTS','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
    # df_valid_mod['text'] = df_valid_mod.QUESTION +". "+ df_valid_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question before
    # # df_train_mod['text'] = df_valid_mod.CONTEXTS.apply(lambda x : (' ').join(x)) +" "+ df_valid_mod.QUESTION  #question after
    # df_valid_mod.drop(columns=['QUESTION','CONTEXTS'],inplace=True)
    # df_valid_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']

    # df_test_mod =
df_test[['QUESTION','CONTEXTS','final_decision','reasoning_required_pred','reasoning_free_pred']].reset_index().copy()
    # df_test_mod['text'] = df_test_mod.QUESTION +". "+ df_test_mod.CONTEXTS.apply(lambda x : (' ').join(x)) #question before
    # # df_test_mod['text'] = df_test_mod.CONTEXTS.apply(lambda x : (' ').join(x)) +" "+ df_test_mod.QUESTION  #question after
    # df_test_mod.drop(columns=['QUESTION','CONTEXTS'],inplace=True)
    # df_test_mod.columns = ['id','label','reasoning_required_pred','reasoning_free_pred','text']

    # id2label={ 0: "no", 1: "maybe", 2: "yes"}
    # label2id = dict((v,k) for k,v in id2label.items())
    # df_train_mod['label'] = df_train_mod.label.apply(lambda x : label2id[x])
    # df_valid_mod['label'] = df_valid_mod.label.apply(lambda x : label2id[x])
    # df_test_mod['label'] = df_test_mod.label.apply(lambda x : label2id[x])

    dataset = DatasetDict({
        'train':Dataset.from_pandas(df_train),
        'validation':Dataset.from_pandas(df_valid),
        'test':Dataset.from_pandas(df_test),
    })

    dataset.remove_columns_(['CONTEXTS','LABELS', 'MESHES','YEAR', 'reasoning_required_pred', 'reasoning_free_pred',
'LONG_ANSWER','__index_level_0__'])
    dataset.rename_column_("final_decision", "labels")
    return dataset.class_encode_column("labels")

dataset = get_pubmedqa_fold(list_data_fold,df_test,0)
dataset.num_rows

"""The `dataset` object itself is [`DatasetDict`](https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasetdict),
which contains one key for the training, validation and test set (with more keys for the mismatched validation and test set in the special
case of `mnli`).

To access an actual element, you need to select a split first, then give an index:
"""

dataset["train"].features

dataset["train"][0]

"""To get a sense of what the data looks like, the following function will show some examples picked randomly in the dataset."""

import datasets
import random
```

```
import pandas as pd
from IPython.display import display, HTML

def show_random_elements(dataset, num_examples=10):
    assert num_examples <= len(dataset), "Can't pick more elements than there are in the dataset."
    picks = []
    for _ in range(num_examples):
        pick = random.randint(0, len(dataset)-1)
        while pick in picks:
            pick = random.randint(0, len(dataset)-1)
        picks.append(pick)

    df = pd.DataFrame(dataset[picks])
    for column, typ in dataset.features.items():
        if isinstance(typ, datasets.ClassLabel):
            df[column] = df[column].transform(lambda i: typ.names[i])
    display(HTML(df.to_html()))

# show_random_elements(dataset["train"])

"""The metric is an instance of
[`datasets.Metric`](https://huggingface.co/docs/datasets/package_reference/main_classes.html#datasets.Metric):"""

# https://github.com/huggingface/datasets/tree/master/metrics
from datasets import load_metric
metric = load_metric("accuracy")

# metric = load_metric("https://github.com/huggingface/datasets/blob/master/metrics/accuracy/accuracy.py")
# Example of typical usage
# for batch in dataset:
#     inputs, references = batch
#     predictions = model(inputs)
#     metric.add_batch(predictions=predictions, references=references)
# score = metric.compute()

metric

"""You can call its `compute` method with your predictions and labels directly and it will return a dictionary with the metric(s) value:"""

import numpy as np

fake_preds = np.random.randint(0, 3, size=(64,))
fake_labels = np.random.randint(0, 3, size=(64,))
metric.compute(predictions=fake_preds, references=fake_labels)

"""## Fine-tuning the model"""

def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    # predictions = predictions[:, 0]
    predictions = np.argmax(predictions, axis=1)
    return metric.compute(predictions=predictions, references=labels)

"""#### For all folds"""

import torch

task = "pubmedqa"
# model_checkpoint = "microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext"
# model_checkpoint = "dmis-lab/biobert-base-cased-v1.1"
# model_checkpoint = "bert-base-uncased"

# LEARNING_RATE = 1e-5
# BATCH_SIZE = 8
# NUM_LABELS = 3
MAX_LENGTH = 512

# metric_name = "accuracy"
# model_name = model_checkpoint.split("/")[-1]

from transformers import TrainingArguments, Trainer, EvalPrediction

def compute_accuracy(p: EvalPrediction):
    preds = np.argmax(p.predictions, axis=1)
```

```
    return {"acc": (preds == p.label_ids).mean()}

# tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)
def preprocess_function(tokenizer,examples):
    return tokenizer(examples["QUESTION"], examples["CONTEXTS_JOINED"],
truncation=True,padding=True,max_length=MAX_LENGTH)



from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score, confusion_matrix, matthews_corrcoef
from sklearn.metrics import precision_recall_fscore_support,accuracy_score

# r, p, f1, _ = my_evaluator.eval_hoc(output_df)

# !pip install --upgrade pyyaml

def train_adapter_pubmedqa(LANG_MODEL,LEARNING_RATE,BATCH_SIZE=8):
    model_name = LANG_MODEL.split("/")[-1]
    tokenizer = AutoTokenizer.from_pretrained(LANG_MODEL, use_fast=True)

    training_args = TrainingArguments(
        learning_rate=LEARNING_RATE,
        num_train_epochs=6,
        per_device_train_batch_size=BATCH_SIZE,
        per_device_eval_batch_size=BATCH_SIZE,
        eval_accumulation_steps=1,
        logging_steps=100,
        output_dir="./training_output",
        overwrite_output_dir=True,
        # The next line is important to ensure the dataset labels are properly passed to the model
        remove_unused_columns=False,
    )

    list_acc = []
    for i in range(10):
        torch.cuda.empty_cache()
        print("###### FOLD",i,"######")
        dataset = get_pubmedqa_fold(list_data_fold,df_test,i)
        print("num_rows",dataset.num_rows)

        encoded_dataset = dataset.map(lambda example: preprocess_function(tokenizer,example), batched=True)
        encoded_dataset.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])

        print("Setup model",LANG_MODEL)
        model = AutoModelForSequenceClassification.from_pretrained(LANG_MODEL, num_labels=3)

        # config = AutoConfig.from_pretrained(
        #     LANG_MODEL,
        #     num_labels=3,
        # )
        # model = AutoModelWithHeads.from_pretrained(
        #     LANG_MODEL,
        #     config=config,
        # )

        # # Add a new adapter
        # model.add_adapter("pubmedqa")
        # # Add a matching classification head
        # model.add_classification_head(
        #     "pubmedqa",
        #     num_labels=3,
        #     # id2label={ 0: "👎", 1: "👍"}
        #   )
        # # Activate the adapter
        # model.train_adapter("pubmedqa")

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

```
        trainer.train()
        output = trainer.evaluate(encoded_dataset['test'])
        print(output)
        list_acc.append(output['eval_acc'])

    print("MEAN ACC",np.mean(list_acc))

for model in ["microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext"]:
    for lr in [1e-5,3e-3,5e-5]:
        train_adapter_pubmedqa(model,lr)

list_acc = []
for i in range(10):
    torch.cuda.empty_cache()
    print("###### FOLD",i,"######")
    dataset = get_pubmedqa_fold(list_data_fold,df_test,i)
    print("num_rows",dataset.num_rows)

    encoded_dataset = dataset.map(preprocess_function, batched=True)
    encoded_dataset.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])

    model = AutoModelForSequenceClassification.from_pretrained(model_checkpoint, num_labels=NUM_LABELS)
    model.add_adapter("pubmedqa")
    model.train_adapter("pubmedqa")

    args = TrainingArguments(
        "test-blurb-adapter-biobert-uncased",
        evaluation_strategy = "epoch",
        save_strategy = "epoch",
        learning_rate=LEARNING_RATE,
        per_device_train_batch_size=BATCH_SIZE,
        per_device_eval_batch_size=BATCH_SIZE,
        eval_accumulation_steps=1,
        num_train_epochs=10,
        weight_decay=0.01,
        load_best_model_at_end=True,
        metric_for_best_model=metric_name,
        # The next line is important to ensure the dataset labels are properly passed to the model
        remove_unused_columns=True,
        # push_to_hub=True,
        # push_to_hub_model_id=f"{model_name}-finetuned-{task}-adapter",
    )

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

    trainer.train()

    commit_msg = ""
    commit_msg += "BATCH_SIZE="+str(BATCH_SIZE)+"\n"
    commit_msg += "LEARNING_RATE="+str(LEARNING_RATE)+"\n"
    commit_msg += "MAX_LENGTH="+str(MAX_LENGTH)+"\n"
    commit_msg += "FOLD="+str(i)+"\n"

    # trainer.push_to_hub(commit_msg)
    output = trainer.evaluate(encoded_dataset['test'])
    list_acc.append(output['eval_accuracy'])

list_acc

np.mean(list_acc)
```