3_1_Lexical_Complexity_Binary_Classification_Prediction_Transformers_

April 7, 2025

[32]:
```
#@title Install Packages
```

[ ]:
```
!pip install -q transformers
!pip install -q torchinfo
!pip install -q datasets
!pip install -q evaluate
!pip install -q nltk
!pip install -q contractions
!pip install -q hf_xet
!pip install -q sentencepiece
```

```
Traceback (most recent call last):
  File "/usr/local/lib/python3.11/dist-
packages/pip/_internal/cli/base_command.py", line 179, in exc_logging_wrapper
    status = run_func(*args)
             ^^^^^^^^^^^^^^^

  File "/usr/local/lib/python3.11/dist-
packages/pip/_internal/cli/req_command.py", line 67, in wrapper
    return func(self, options, args)
           ^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/usr/local/lib/python3.11/dist-
packages/pip/_internal/commands/install.py", line 447, in run
^C
^C
^C
^C
```

[ ]:
```
!sudo apt-get update
! sudo apt-get install tree
```

[ ]:
```
#@title Imports
import nltk
from nltk.tokenize import RegexpTokenizer

import contractions

import evaluate
```

```
import transformers
import torch

from torchinfo import summary

from datasets import load_dataset, Dataset, DatasetDict

from transformers import AutoTokenizer, AutoModel,␣
 ↪AutoModelForSequenceClassification, TrainingArguments, Trainer, BertConfig,␣
 ↪BertForSequenceClassification

import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import sklearn

import spacy

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report,␣
 ↪precision_recall_fscore_support, accuracy_score

import sentencepiece

from datetime import datetime
```

[ ]:
```
# @title Mount Google Drive
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:
```
dir_root = '/content/drive/MyDrive/266-final/'
# dir_data = '/content/drive/MyDrive/266-final/data/'
# dir_data = '/content/drive/MyDrive/266-final/data/se21-t1-comp-lex-master/'
dir_data = '/content/drive/MyDrive/266-final/data/266-comp-lex-master'
dir_models = '/content/drive/MyDrive/266-final/models/'
dir_results = '/content/drive/MyDrive/266-final/results/'
```

[ ]:
```
wandbai_api_key = "5236444b7e96f5cf74038116d8c1efba161a4310"
```

[ ]:
```
!tree /content/drive/MyDrive/266-final/data/266-comp-lex-master/
```

[ ]:
```
!ls -R /content/drive/MyDrive/266-final/data/266-comp-lex-master/
```

[ ]:
```
!tree /content/drive/MyDrive/266-final/data/266-comp-lex-master/
```

[ ]:
```
#@title Import Data
```

[ ]:
```
df_names = [
    "train_single_df",
    "train_multi_df",
    "trial_val_single_df",
    "trial_val_multi_df",
    "test_single_df",
    "test_multi_df"
]

loaded_dataframes = {}

for df_name in df_names:
    if "train" in df_name:
        subdir = "fe-train"
    elif "trial_val" in df_name:
        subdir = "fe-trial-val"
    elif "test" in df_name:
        subdir = "fe-test-labels"
    else:
        subdir = None

    if subdir:
        read_path = os.path.join(dir_data, subdir, f"{df_name}.csv")
        loaded_df = pd.read_csv(read_path)
        loaded_dataframes[df_name] = loaded_df
        print(f"Loaded {df_name} from {read_path}")

# for df_name, df in loaded_dataframes.items():
#     print(f"\n>>> {df_name} shape: {df.shape}")
#     if 'binary_complexity' in df.columns:
#         print(df['binary_complexity'].value_counts())
#         print(df.info())
#         print(df.head())

for df_name, df in loaded_dataframes.items():
    globals()[df_name] = df
    print(f"{df_name} loaded into global namespace.")
```

- Functional tests pass, we can proceed with Baseline Modeling

0.1 Experiments with Transformers Models

[ ]:
```
# def get_model_and_tokenizer(model_name: str):
#     """
#     Loads the specified pretrained model & tokenizer for classification.
#     """
#     tokenizer = AutoTokenizer.from_pretrained(model_name)
#     model = AutoModelForSequenceClassification.from_pretrained(model_name)
#     return model, tokenizer

# new prod version to support local model checkpoints, to be used after␣
 ↪experiment 1.0
def get_model_and_tokenizer(
    remote_model_name: str = None,
    local_model_path: str = None
):
    """
    Loads the model & tokenizer for classification.
    If 'local_model_path' is specified, load from that path.
    Otherwise, fall back to 'remote_model_name'.
    """
    from transformers import AutoTokenizer, AutoModelForSequenceClassification

    if local_model_path:
        # Local load
        print(f"Loading from local path: {local_model_path}")
        tokenizer = AutoTokenizer.from_pretrained(local_model_path)
        model = AutoModelForSequenceClassification.
 ↪from_pretrained(local_model_path)
    elif remote_model_name:
        # Load from HF Hub
        print(f"Loading from Hugging Face model: {remote_model_name}")
        tokenizer = AutoTokenizer.from_pretrained(remote_model_name)
        model = AutoModelForSequenceClassification.
 ↪from_pretrained(remote_model_name)
    else:
        raise ValueError("You must provide either a remote_model_name or a␣
 ↪local_model_path!")

    return model, tokenizer
```

[ ]:
```
def freeze_unfreeze_layers(model, layers_to_unfreeze=None):
    """
    Toggles requires_grad = False for all parameters
    except for those whose names contain any string in layers_to_unfreeze.
    By default, always unfreeze classifier/heads.
    """
```

```
    if layers_to_unfreeze is None:
        layers_to_unfreeze = ["classifier.", "pooler."]

    for name, param in model.named_parameters():
        # If any layer substring matches, we unfreeze
        if any(substring in name for substring in layers_to_unfreeze):
            param.requires_grad = True
        else:
            param.requires_grad = False
```

[ ]:
```
def encode_examples(examples, tokenizer, text_col, max_length=256):
    """
    Tokenizes a batch of texts from 'examples[text_col]' using the given␣
 ↪tokenizer.
    Returns a dict with 'input_ids', 'attention_mask', etc.
    """
    texts = examples[text_col]
    encoded = tokenizer(
        texts,
        truncation=True,
        padding='max_length',
        max_length=max_length
    )
    return encoded
```

[ ]:
```
def prepare_dataset(df, tokenizer, text_col, label_col, max_length=256):
    """
    Converts a Pandas DataFrame to a Hugging Face Dataset,
    then applies 'encode_examples' to tokenize.
    """
    # Convert to HF Dataset
    dataset = Dataset.from_pandas(df)

    # Map the encode function
    dataset = dataset.map(
        lambda batch: encode_examples(batch, tokenizer, text_col, max_length),
        batched=True
    )

    # Rename the label column to 'labels' for HF Trainer
    dataset = dataset.rename_column(label_col, "labels")
    # HF often requires removing any columns that cannot be converted or are␣
 ↪not needed
    dataset.set_format(type='torch',
                       columns=['input_ids', 'attention_mask', 'labels'])
    return dataset
```

[ ]:
```
def compute_metrics(eval_pred):
    """
    Computes classification metrics, including accuracy, precision, recall, and␣
 ↪F1.
    """
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=1)

    metric_accuracy  = evaluate.load("accuracy")
    metric_precision = evaluate.load("precision")
    metric_recall    = evaluate.load("recall")
    metric_f1        = evaluate.load("f1")

    accuracy_result  = metric_accuracy.compute(predictions=preds,␣
 ↪references=labels)
    precision_result = metric_precision.compute(predictions=preds,␣
 ↪references=labels, average="binary")
    recall_result    = metric_recall.compute(predictions=preds,␣
 ↪references=labels, average="binary")
    f1_result        = metric_f1.compute(predictions=preds, references=labels,␣
 ↪average="binary")

    return {
        "accuracy"       : accuracy_result["accuracy"],
        "precision": precision_result["precision"],
        "recall"   : recall_result["recall"],
        "f1"       : f1_result["f1"]
    }
```

0.1.1 Experiment Design

[ ]:
```
# Define Experiment Parameters

named_model = "bert-base-cased"
# named_model = "roberta-base"
# named_model = "bert-large"
# named_model = "roberta-large"
# named_model = "" # modern bert

# learning_rate = 1e-3
# learning_rate = 1e-4
# learning_rate = 1e-5
# learning_rate = 5e-6
learning_rate = 5e-7
# learning_rate = 5e-8

# num_epochs = 3
```

```
num_epochs = 5
# num_epochs = 10
# num_epochs = 15
# num_epochs = 20

length_max = 128
# length_max = 256
# length_max = 348
# length_max = 512

# size_batch = 1
# size_batch = 4
size_batch = 8
# size_batch = 16
# size_batch = 24
# size_batch = 32

regularization_weight_decay = 0
# regularization_weight_decay = 0.1
# regularization_weight_decay = 0.5

# dropout???

# layers to freeze and unfreeze?

y_col = "binary_complexity"
# y_col = "complexity"

x_task = "single"
# x_task = "multi"

# x_col = "sentence"
x_col = "sentence_no_contractions"
# x_col = "pos_sequence"
# x_col = "dep_sequence"
# x_col = "morph_sequence"

if x_task == "single":
    df_train = train_single_df
    df_val   = trial_val_single_df
    df_test  = test_single_df
else:
    df_train = train_multi_df
    df_val   = trial_val_multi_df
    df_test  = test_multi_df
```

[ ]:
```python
def train_transformer_model(
    model,
    tokenizer,
    train_dataset,
    val_dataset,
    output_dir=dir_results,
    num_epochs=num_epochs,
    batch_size=size_batch,
    lr=learning_rate,
    weight_decay=regularization_weight_decay
):
    """
    Sets up a Trainer and trains the model for 'num_epochs' using the given␣
 ↪dataset.
    Returns the trained model and the Trainer object for possible re-use or␣
 ↪analysis.
    """

    training_args = TrainingArguments(
        output_dir=output_dir,
        num_train_epochs=num_epochs,
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        evaluation_strategy="epoch",
        save_strategy="no",
        logging_strategy="epoch",
        learning_rate=lr,
        weight_decay=weight_decay,
        report_to=["none"],  # or "wandb"
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        tokenizer=tokenizer,  # optional
        compute_metrics=compute_metrics
    )

    trainer.train()
    return model, trainer
```

0.1.2 1.0: from pretrained bert-base-cased single task 1

Model Inspection

```
[ ]: print("model checkpoints:", dir_models)
     !ls /content/drive/MyDrive/266-final/models/
```

```
[ ]: # Load Model & Tokenizer
     # model, tokenizer = get_model_and_tokenizer(named_model) # deprecated argument␣
      ↪structure
     model, tokenizer = get_model_and_tokenizer("/content/drive/MyDrive/266-final/
      ↪models/....") # proposed argument usage for checkpointed models

     for name, param in model.named_parameters():
         print(name)

     print("=============")
     print(named_model, ":")
     print("=============")
     print(model)
     print("=============")
     print(model.config)
     print("=============")
     print("num_parameters:", model.num_parameters())
     print("=============")
     print("num_trainable_parameters:", model.num_parameters(only_trainable=True))
```

Layer Configuration

```
[ ]: # Freeze/Unfreeze Layers & Additional Configuration Parameters

     import torch.nn as nn

     layers_to_unfreeze = [
         "bert.encoder.layer.9.",
         "bert.encoder.layer.10.",
         "bert.encoder.layer.11.",
         "pooler.",
         "classifier.",
     ]

     freeze_unfreeze_layers(model, layers_to_unfreeze=layers_to_unfreeze)


     bert_config = BertConfig(
         # vocab_size=28996,
         hidden_size=768,
         # num_hidden_layers=12,
         # num_attention_heads=12,
         # intermediate_size=3072,
         intermediate_size=6144,
         # max_position_embeddings=512,
```

```
    type_vocab_size=2,

    hidden_dropout_prob=0.1,
    attention_probs_dropout_prob=0.1,
    # classifier_dropout=None,
    # initializer_range=0.02,
    # layer_norm_eps=1e-12,

    hidden_act="gelu",
    gradient_checkpointing=True,
    position_embedding_type="absolute",
    use_cache=True,
    pad_token_id=0
)

model.bert.pooler.activation = nn.ReLU() # Tanh() replaced as the pooler layer␣
 ↪activation function

for name, param in model.named_parameters():
    print(name, "requires_grad=", param.requires_grad)

print("\nLayers that are 'True' are trainable. 'False' are frozen.")

print("==============")
print(named_model, ":")
print("==============")
print(model)
print("==============")
print(model.config)
print("==============")
print("num_parameters:", model.num_parameters())
print("==============")
print("num_trainable_parameters:", model.num_parameters(only_trainable=True))
```

Dataset Preparation

[ ]:
```
# Tokenize & Prepare Datasets

train_data_hf = prepare_dataset(
    df_train,
    tokenizer,
    text_col=x_col,
    label_col=y_col,
    max_length=length_max
)

val_data_hf = prepare_dataset(
    df_val,
```

```
    tokenizer,
    text_col=x_col,
    label_col=y_col,
    max_length=length_max
)

test_data_hf = prepare_dataset(
    df_test,
    tokenizer,
    text_col=x_col,
    label_col=y_col,
    max_length=length_max

)

print("Datasets prepared. Sample from train_data_hf:\n", train_data_hf[10])
print("Datasets prepared. Sample from train_data_hf:\n", val_data_hf[10])
print("Datasets prepared. Sample from train_data_hf:\n", test_data_hf[10])
```

1.0 Results

[ ]:
```
# Train & Evaluate

trained_model, trainer_obj = train_transformer_model(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_data_hf,
    val_dataset=val_data_hf,
    output_dir=dir_results,
    num_epochs=num_epochs,
    batch_size=size_batch,
    lr=learning_rate,
    weight_decay=regularization_weight_decay
)

metrics = trainer_obj.evaluate()
print("Validation metrics:", metrics)

test_metrics = trainer_obj.evaluate(test_data_hf) if test_data_hf else None
print("Test metrics:", test_metrics)
```

[ ]:
```
print("Experiment configuration used with this experiment:")
print("model used:", named_model)
print("learning rate used:", learning_rate)
print("number of epochs:", num_epochs)
print("maximum sequence length:", length_max)
print("batch size used:", size_batch)
print("regularization value:", regularization_weight_decay)
```

```
print("outcome variable:", y_col)
print("task:", x_task)
print("input column:", x_col)
```

[ ]:
```
# save model checkpoint

timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
model_save_path = os.path.join(dir_models,␣
  ↪f"{x_task}_{named_model}_{y_col}_{timestamp}")

trainer_obj.save_model(model_save_path)
print(f"Model checkpoint saved to: {model_save_path}")
```

0.1.3 Experiment 1.1: from checkpoint bert-base-cased single task 1

[ ]:
```
# Define Experiment Parameters

named_model = "bert-base-cased"
# named_model = "roberta-base"
# named_model = "bert-large"
# named_model = "roberta-large"
# named_model = "" # modern bert

# learning_rate = 1e-3
# learning_rate = 1e-4
# learning_rate = 1e-5
# learning_rate = 5e-6
# learning_rate = 5e-7
learning_rate = 1e-8

# num_epochs = 3
num_epochs = 5
# num_epochs = 10
# num_epochs = 15
# num_epochs = 20

length_max = 128
# length_max = 256
# length_max = 348
# length_max = 512

# size_batch = 1
size_batch = 4
# size_batch = 8
# size_batch = 16
# size_batch = 24
# size_batch = 32
```

```
# regularization_weight_decay = 0
regularization_weight_decay = 0.1
# regularization_weight_decay = 0.5

y_col = "binary_complexity"
# y_col = "complexity"

x_task = "single"
# x_task = "multi"

# x_col = "sentence"
x_col = "sentence_no_contractions"
# x_col = "pos_sequence"
# x_col = "dep_sequence"
# x_col = "morph_sequence"

if x_task == "single":
    df_train = train_single_df
    df_val   = trial_val_single_df
    df_test  = test_single_df
else:
    df_train = train_multi_df
    df_val   = trial_val_multi_df
    df_test  = test_multi_df
```

[ ]:
```
# Load Model & Tokenizer
model, tokenizer = get_model_and_tokenizer(named_model) # deprecated argument␣
 ↪structure
# model, tokenizer = get_model_and_tokenizer("/content/drive/MyDrive/266-final/
 ↪models/bert-base-cased_20250407_232900") # proposed argument usage for␣
 ↪checkpointed models

# for name, param in model.named_parameters():
#     print(name)

print("==============")
print(named_model, ":")
print("==============")
# print(model)
print("==============")
print(model.config)
# print("==============")
# print("num_parameters:", model.num_parameters())
# print("==============")
# print("num_trainable_parameters:", model.num_parameters(only_trainable=True))
```

[ ]:
```
# Freeze/Unfreeze Layers & Additional Configuration Parameters

import torch.nn as nn

layers_to_unfreeze = [
    "bert.embeddings.",
    "bert.encoder.layer.0.",
    "bert.encoder.layer.1.",
    "bert.encoder.layer.9.",
    "bert.encoder.layer.9.",
    "bert.encoder.layer.10.",
    "bert.encoder.layer.11.",
    "bert.pooler.",
    "classifier.",
]

freeze_unfreeze_layers(model, layers_to_unfreeze=layers_to_unfreeze)


bert_config = BertConfig(
    # vocab_size=28996,
    hidden_size=768,
    # num_hidden_layers=12,
    # num_attention_heads=12,
    intermediate_size=6144,
    # max_position_embeddings=512,
    type_vocab_size=2,

    hidden_dropout_prob=0.1,
    attention_probs_dropout_prob=0.1,
    # classifier_dropout=None,
    # initializer_range=0.02,
    # layer_norm_eps=1e-12,

    hidden_act="gelu",
    gradient_checkpointing=True,
    position_embedding_type="absolute",
    use_cache=True,
    pad_token_id=0
)

model.bert.pooler.activation = nn.ReLU() # Tanh() replaced as the pooler layer␣
 ↪activation function

for name, param in model.named_parameters():
    print(name, "requires_grad=", param.requires_grad)
```

```
print("\nLayers that are 'True' are trainable. 'False' are frozen.")

print("=============")
print(named_model, ":")
print("=============")
print(model)
print("=============")
print(model.config)
print("=============")
print("num_parameters:", model.num_parameters())
print("=============")
print("num_trainable_parameters:", model.num_parameters(only_trainable=True))
```

1.1 Results

[ ]:
```
# Train & Evaluate

trained_model, trainer_obj = train_transformer_model(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_data_hf,
    val_dataset=val_data_hf,
    output_dir=dir_results,
    num_epochs=num_epochs,
    batch_size=size_batch,
    lr=learning_rate,
    weight_decay=regularization_weight_decay
)

metrics = trainer_obj.evaluate()
print("Validation metrics:", metrics)

test_metrics = trainer_obj.evaluate(test_data_hf) if test_data_hf else None
print("Test metrics:", test_metrics)
```

[ ]:
```
print("Experiment configuration used with this experiment:")
print("model used:", named_model)
print("learning rate used:", learning_rate)
print("number of epochs:", num_epochs)
print("maximum sequence length:", length_max)
print("batch size used:", size_batch)
print("regularization value:", regularization_weight_decay)
print("outcome variable:", y_col)
print("task:", x_task)
print("input column:", x_col)
```

[ ]:
```
# save model checkpoint
```

```
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
model_save_path = os.path.join(dir_models,␣
 ↪f"{x_task}_{named_model}_{y_col}_{timestamp}")

trainer_obj.save_model(model_save_path)
print(f"Model checkpoint saved to: {model_save_path}")
```

0.1.4 Experiment 1.2: from pre-trained bert-base-cased multi task 2

[ ]:
```
# Define Experiment Parameters

named_model = "bert-base-cased"
# named_model = "roberta-base"
# named_model = "bert-large"
# named_model = "roberta-large"
# named_model = "" # modern bert

# learning_rate = 1e-3
# learning_rate = 1e-4
# learning_rate = 1e-5
# learning_rate = 5e-6
# learning_rate = 5e-7
learning_rate = 1e-8

# num_epochs = 3
num_epochs = 5
# num_epochs = 10
# num_epochs = 15
# num_epochs = 20

length_max = 128
# length_max = 256
# length_max = 348
# length_max = 512

# size_batch = 1
size_batch = 4
# size_batch = 8
# size_batch = 16
# size_batch = 24
# size_batch = 32

# regularization_weight_decay = 0
regularization_weight_decay = 0.1
# regularization_weight_decay = 0.5

y_col = "binary_complexity"
```

```
# y_col = "complexity"

# x_task = "single"
x_task = "multi"

# x_col = "sentence"
x_col = "sentence_no_contractions"
# x_col = "pos_sequence"
# x_col = "dep_sequence"
# x_col = "morph_sequence"

if x_task == "single":
    df_train = train_single_df
    df_val   = trial_val_single_df
    df_test  = test_single_df
else:
    df_train = train_multi_df
    df_val   = trial_val_multi_df
    df_test  = test_multi_df
```

[ ]:
```
print("model checkpoints:", dir_models)
!ls /content/drive/MyDrive/266-final/models/
```

[ ]:
```
# Load Model & Tokenizer
model, tokenizer = get_model_and_tokenizer(named_model) # deprecated argument␣
 ↪structure
# model, tokenizer = get_model_and_tokenizer("/content/drive/MyDrive/266-final/
 ↪models/bert-base-cased_20250407_232900") # proposed argument usage for␣
 ↪checkpointed models

# for name, param in model.named_parameters():
#     print(name)

print("==============")
print(named_model, ":")
print("==============")
# print(model)
print("==============")
print(model.config)
# print("==============")
# print("num_parameters:", model.num_parameters())
# print("==============")
# print("num_trainable_parameters:", model.num_parameters(only_trainable=True))
```

[ ]:
```
# Freeze/Unfreeze Layers & Additional Configuration Parameters

import torch.nn as nn
```

```
layers_to_unfreeze = [
    "bert.embeddings.",
    "bert.encoder.layer.0.",
    "bert.encoder.layer.1.",
    "bert.encoder.layer.9.",
    "bert.encoder.layer.9.",
    "bert.encoder.layer.10.",
    "bert.encoder.layer.11.",
    "bert.pooler.",
    "classifier.",
]

freeze_unfreeze_layers(model, layers_to_unfreeze=layers_to_unfreeze)


bert_config = BertConfig(
    # vocab_size=28996,
    hidden_size=768,
    # num_hidden_layers=12,
    # num_attention_heads=12,
    intermediate_size=6144,
    # max_position_embeddings=512,
    type_vocab_size=2,

    hidden_dropout_prob=0.1,
    attention_probs_dropout_prob=0.1,
    # classifier_dropout=None,
    # initializer_range=0.02,
    # layer_norm_eps=1e-12,

    hidden_act="gelu",
    gradient_checkpointing=True,
    position_embedding_type="absolute",
    use_cache=True,
    pad_token_id=0
)

model.bert.pooler.activation = nn.ReLU() # Tanh() replaced as the pooler layer␣
 ↪activation function

for name, param in model.named_parameters():
    print(name, "requires_grad=", param.requires_grad)

print("\nLayers that are 'True' are trainable. 'False' are frozen.")

print("==============")
```

```
print(named_model, ":")
print("=============")
print(model)
print("=============")
print(model.config)
print("=============")
print("num_parameters:", model.num_parameters())
print("=============")
print("num_trainable_parameters:", model.num_parameters(only_trainable=True))
```

1.2 Results

[ ]:
```
# Train & Evaluate

trained_model, trainer_obj = train_transformer_model(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_data_hf,
    val_dataset=val_data_hf,
    output_dir=dir_results,
    num_epochs=num_epochs,
    batch_size=size_batch,
    lr=learning_rate,
    weight_decay=regularization_weight_decay
)

metrics = trainer_obj.evaluate()
print("Validation metrics:", metrics)

test_metrics = trainer_obj.evaluate(test_data_hf) if test_data_hf else None
print("Test metrics:", test_metrics)
```

[ ]:
```
print("Experiment configuration used with this experiment:")
print("model used:", named_model)
print("learning rate used:", learning_rate)
print("number of epochs:", num_epochs)
print("maximum sequence length:", length_max)
print("batch size used:", size_batch)
print("regularization value:", regularization_weight_decay)
print("outcome variable:", y_col)
print("task:", x_task)
print("input column:", x_col)
```

[ ]:
```
# save model checkpoint

timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
model_save_path = os.path.join(dir_models,␣
 ↪f"{x_task}_{named_model}_{y_col}_{timestamp}")
```

```
trainer_obj.save_model(model_save_path)
print(f"Model checkpoint saved to: {model_save_path}")
```

0.1.5 Experiment 1.3: from checkpoint 1.0 or 1.1 bert-base-cased MSFT'd single -> multi

[ ]:
```
# Define Experiment Parameters

named_model = "bert-base-cased"
# named_model = "roberta-base"
# named_model = "bert-large"
# named_model = "roberta-large"
# named_model = "" # modern bert

# learning_rate = 1e-3
# learning_rate = 1e-4
# learning_rate = 1e-5
# learning_rate = 5e-6
# learning_rate = 5e-7
learning_rate = 1e-8

# num_epochs = 3
num_epochs = 5
# num_epochs = 10
# num_epochs = 15
# num_epochs = 20

length_max = 128
# length_max = 256
# length_max = 348
# length_max = 512

# size_batch = 1
size_batch = 4
# size_batch = 8
# size_batch = 16
# size_batch = 24
# size_batch = 32

# regularization_weight_decay = 0
regularization_weight_decay = 0.1
# regularization_weight_decay = 0.5

y_col = "binary_complexity"
# y_col = "complexity"
```

```
# x_task = "single"
x_task = "multi"

# x_col = "sentence"
x_col = "sentence_no_contractions"
# x_col = "pos_sequence"
# x_col = "dep_sequence"
# x_col = "morph_sequence"

if x_task == "single":
    df_train = train_single_df
    df_val   = trial_val_single_df
    df_test  = test_single_df
else:
    df_train = train_multi_df
    df_val   = trial_val_multi_df
    df_test  = test_multi_df
```

[ ]:
```
print("model checkpoints:", dir_models)
!ls /content/drive/MyDrive/266-final/models/
```

[ ]:
```
# Load Model & Tokenizer
# model, tokenizer = get_model_and_tokenizer(named_model) # deprecated argument structure
model, tokenizer = get_model_and_tokenizer("/content/drive/MyDrive/266-final/models/....") # proposed argument usage for checkpointed models

# for name, param in model.named_parameters():
#     print(name)

print("=============")
print(named_model, ":")
print("=============")
# print(model)
print("=============")
print(model.config)
# print("=============")
# print("num_parameters:", model.num_parameters())
# print("=============")
# print("num_trainable_parameters:", model.num_parameters(only_trainable=True))
```

[ ]:
```
# Freeze/Unfreeze Layers & Additional Configuration Parameters

import torch.nn as nn

layers_to_unfreeze = [
    "bert.embeddings.",
```

```
    "bert.encoder.layer.0.",
    "bert.encoder.layer.1.",
    "bert.encoder.layer.9.",
    "bert.encoder.layer.9.",
    "bert.encoder.layer.10.",
    "bert.encoder.layer.11.",
    "bert.pooler.",
    "classifier.",
]

freeze_unfreeze_layers(model, layers_to_unfreeze=layers_to_unfreeze)


bert_config = BertConfig(
    # vocab_size=28996,
    hidden_size=768,
    # num_hidden_layers=12,
    # num_attention_heads=12,
    intermediate_size=6144,
    # max_position_embeddings=512,
    type_vocab_size=2,

    hidden_dropout_prob=0.1,
    attention_probs_dropout_prob=0.1,
    # classifier_dropout=None,
    # initializer_range=0.02,
    # layer_norm_eps=1e-12,

    hidden_act="gelu",
    gradient_checkpointing=True,
    position_embedding_type="absolute",
    use_cache=True,
    pad_token_id=0
)

model.bert.pooler.activation = nn.ReLU() # Tanh() replaced as the pooler layer␣
 ↪activation function

for name, param in model.named_parameters():
    print(name, "requires_grad=", param.requires_grad)

print("\nLayers that are 'True' are trainable. 'False' are frozen.")

print("=============")
print(named_model, ":")
print("=============")
print(model)
```

```
print("==============")
print(model.config)
print("==============")
print("num_parameters:", model.num_parameters())
print("==============")
print("num_trainable_parameters:", model.num_parameters(only_trainable=True))
```

1.3 Results

[ ]:
```
# Train & Evaluate

trained_model, trainer_obj = train_transformer_model(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_data_hf,
    val_dataset=val_data_hf,
    output_dir=dir_results,
    num_epochs=num_epochs,
    batch_size=size_batch,
    lr=learning_rate,
    weight_decay=regularization_weight_decay
)

metrics = trainer_obj.evaluate()
print("Validation metrics:", metrics)

test_metrics = trainer_obj.evaluate(test_data_hf) if test_data_hf else None
print("Test metrics:", test_metrics)
```

[ ]:
```
print("Experiment configuration used with this experiment:")
print("model used:", named_model)
print("learning rate used:", learning_rate)
print("number of epochs:", num_epochs)
print("maximum sequence length:", length_max)
print("batch size used:", size_batch)
print("regularization value:", regularization_weight_decay)
print("outcome variable:", y_col)
print("task:", x_task)
print("input column:", x_col)
```

[ ]:
```
# save model checkpoint

timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
model_save_path = os.path.join(dir_models,␣
  ↪f"{x_task}_{named_model}_{y_col}_{timestamp}")

trainer_obj.save_model(model_save_path)
print(f"Model checkpoint saved to: {model_save_path}")
```

[ ]: