A Clinical Text Corpus for Emotion and Mental Health Classification Using ClinicalBERT

Ferdinand Blaire C. Animas
Computer Studies and Engineering
Jose Rizal University
Mandaluyong City, Philippines
ferdinandblaire.animas@my.jru.edu

Ivan James C. Hernandez
Computer Studies and Engineering
Jose Rizal University
Mandaluyong City, Philippines
ivanjames.hernandez@my.jru.edu

Alysson Hana A. Perez
Computer Studies and Engineering
Jose Rizal University
Mandaluyong City, Philippines
alyssonhana.perez@my.jru.edu

***Abstract*—Early identification of mental-health indicators from self-reported textual statements remains a critical challenge in clinical practice due to the subjective, non-structured, and emotionally nuanced nature of patient language. This study proposes and evaluates a ClinicalBERT-based fine-tuned model for automated emotion and mental-state classification, with a particular emphasis on detecting stress from short patient-like inputs. Leveraging the domain-specific pretraining of ClinicalBERT on clinical corpora, the system aims to reduce misinterpretation arising from vague or non-medical descriptions of psychological states. A dataset of labeled emotional expressions was preprocessed, tokenized, and used to fine-tune the model under varying hyperparameter configurations conducted by three experimenters. Results showed consistently high model confidence across all experiments (0.9995–0.9996), indicating strong model alignment with stress-related linguistic patterns. Comparative analysis highlighted minor performance differences, with the configuration by Perez achieving the highest stress probability. Overall, the study demonstrates the feasibility and effectiveness of ClinicalBERT in supporting mental-health assessment, aligning with global health priorities and offering a pathway toward scalable, AI-driven decision-support tools for early psychological triage.**

Keywords—ClinicalBERT, emotion classification, mental health, natural language processing (NLP), transformer models, sentiment analysis, Sustainable Development Goal (SDG) 3.

I. INTRODUCTION

Textual data emotion and mental-health identification analysis has been an area of considerable research focus in natural language processing (NLP) and clinical informatics in the past several years. Perceptiveness and contextual contingency of human expression makes emotion recognition in text other than positive/negative sentiment particularly challenging to specific emotion recognition (e.g., anxiety, fear, sadness) [1], [2]. It is this prospective to early intervention, monitoring, and individualized care that is displayed to be promising in the clinical or mental-health contexts is the automatic categorization of the statements made by the patient into such categories as anxiety, confusion, restlessness, or other affective conditions [3].

The study is consistent with the United Nations Sustainable Development Goal (SDG) 3: Good Health and Well-Being, and more specifically with Target 3.4, that states the focus on mental health and well-being, the promotion of which should come at the cost of a reduction in premature deaths caused by non-communicable diseases by prevention and treatment [14]. The suggested ClinicalBERT-based model will fit into this international project by implementing artificial intelligence to support mental health assessment and early detection. The study contributes to the ability to intervene in time and makes clinical decisions with data, as it is based on automated emotion and sentiment detection of patient-like text. AI implementation into mental health analysis is in accordance with the World Health Organization Comprehensive Mental Health Action Plan 20132030, which recommends the use of digital technologies to empower mental health systems and access to services [15]. Moreover, the work can be used to address the call to find scalable, equitable, and sustainable mental health solutions in the whole world since it limits diagnostic bias and enhances detection accuracy [16].

The data set in question consists of short self-reporting data in the form of statements, and it is labeled as one of the statuses (e.g. Anxiety), thus, offering supervised classification data of textual emotional or mental-state identification of text. Since the text has been patient-like or self-reported (as it is not just a structured clinical note), the data falls between the vernacular of mental-health expression, and the vernacular of clinical documentation. One can also learn and test models, which strive to extrapolate the underlying emotion/psychological condition given solely the linguistic clues and such text-label pairs [4].

The BERT architecture based on transformer ClinicalBERT version in the form of the clinical/biomedical text occupies an especially appropriate niche in this assignment. The ClinicalBERT has been trained on large clinical note corpora such as the MIMIC-III dataset and has been useful in clinical language pattern modellings [5], [6]. Indicatively, Huang et al. [7] set ClinicalBERT to be superior among the discharge summary based hospital readmission prediction models. Such cases are a benefit of this domain-specific pre-training whose textual inputs are written in the language of clinical or health-related nature,

and with or without self-understandable or self-report wordings [8].

The use of ClinicalBERT in regards to the given data will allow using pre-trained models of clinical language and further optimize the model retrieved to meet the particular problem of emotional or mental-state classification. The acumen of the clinical and behavioural language of the model can lead to the optimal discrimination over a general sentiment-analysis model because the phrases in the dataset refer to mental-health issues (e.g., trouble sleeping, restless heart, confused mind), which the sentiment-analysis model can carry out [9]. Besides, the fine-tuning paradigm enables the model to self-report the less strict nature of the data, and retain the underlying clinical terminology during pre-training [10].

Regarding the methodological aspect, there are a number of significant things that may be considered when training the dataset to the models. Some preprocessing (tokenization, cleaning, missing values) and balancing of classes might be necessary to reduce the under-represented emotional states. This is because the entries of the text are finite and, therefore, model configuration (sequence length, truncation strategy, batch size) must be configured to the context to maintain the text. Multi-class evaluation measures, like accuracy, F1-score and AUC are used in the evaluation. Such are in correspondence with the available literature on text emotion detection and classification [1], [4], [11].

The dataset may also be applicable to practice in the domain of mental-health surveillance, sentiment surveillance on the population level in a healthcare institution, and automatic text triage of patient-reports. In the digital level of mental-health services, e.g. a trained model based on this data might automatically identify the usages that can denote the manifestation of anxiety or distress to be followed up by a human in a timely manner [12]. The existence of such systems can be useful to clinicians and researchers when it comes to their perception of the mental-health trends of populations and enhancing patient outcomes.

The overall excitement of this information in self-report brief statements involving emotional or psychological condition is better befitting the domain-sensitive model like ClinicalBERT. The work of text that is clinically relevant, labelled emotions, and domain specific modeling provides a solid basis on which the research of the automated sentiment and emotion analysis in mental-health settings can be based. This information may be an invaluable contribution to the production of explainable, clinically intelligent NLP systems where a systematic preprocessing, classification, and reorganization of the models is carried out.

Fine-tuning large pretrained language models offers a principled way to transfer the syntactic and semantic priors learned from massive unlabeled corpora into our specific classification task, yielding large gains over training from scratch or hand-engineered TF–IDF features. In practice, task-specific fine-tuning of Transformer encoders such as BERT consistently improves macro-F1 on text classification benchmarks by aligning the model's contextual representations with the target label space [17], [18]. For code-switched Filipino–English inputs, multilingual encoders like **XLM-RoBERTa** provide robust subword representations and cross-lingual generalization, which have been shown to help downstream sentiment tasks in low-resource or mixed-language settings [19]. Moreover, **domain adaptation**—either via domain-adaptive pretraining or careful task-level fine-tuning—improves robustness to in-domain vocabulary and style shifts [20]. Complementing this, Filipino-prior encoders such as **RoBERTa-Tagalog** use subword vocabularies tailored to Tagalog morphology and orthography, which can reduce out-of-vocabulary effects and stabilize optimization for local-language sentiment datasets [21]. These findings collectively motivate our fine-tuning setup and the inclusion of both multilingual and Filipino-prior backbones in our experiments.

II. Problem Statement

Misdiagnosis or underdiagnosis of mental-health conditions, including anxiety, depression, or other associated emotional disorders, is one of the most important problems in clinical practice. This is a problem in that a clinical assessment is highly dependent on the human subjective interpretation of verbal or written reports of a patient. Patients can talk about their symptoms vaguely, emotionally or non-medically e.g. when someone says I am restless or having trouble sleeping or I am so confused in my head as depicted in the dataset. The absence of systematic tools that analyze these subtle emotional statements may cause healthcare professionals to miss vital clues about the psychological issues, which results in a false or slow diagnosis. The issue is particularly important in the mental health sphere, where language patterns can be used as the first and the most readily available indicator of distress in the background [1], [3], [11].

To cope with it, the project will focus on the creation of the automated sentiment and emotion analysis system with the help of ClinicalBERT, which will be trained on the labeled statements of the dataset. With this model, clinicians might use it as a decision-support tool to determine if the patient-like text includes a pattern of anxiety or emotional disturbance so that there may be an additional, objective analysis [5], [6], [9]. This would not substitute doctors, but this system would act as an aid, as it would point at possible emotional red flags in the stories of patients. Finally, the proposed project aims at alleviating the presence of diagnostic bias, as well as enhancing the capability of the timely detection of mental-health issues by converting the qualitative patient utterances into quantifiable, factual knowledge [7], [8], [10], [12].

III. Research Objectives

The main goal of the project would be to develop a ClinicalBERT based emotion and sentiment analysis model, which would be trained on the labeled clinical-like statements of the dataset comprising of text statements, which would allow healthcare professionals to identify a mental-health condition, such as anxiety, in the initial and correct manner. The project will implement the principles of natural language processing (NLP) and deep learning and convert unstructured patient expressions into measurable indicators of emotions that can be used as a second-level diagnostic instrument among clinicians.

In order to accomplish this general objective, the project will strive to obtain the following specific objectives:

To clean, **tokenize** and encode the text column (statement) and a label column (status) of the data, so that the data can be trained on a model and that there is no noise or inconsistencies in the data.

To optimize the **ClinicalBERT** model on the ready dataset so that it could be capable of classifying the emotional or psychological state based on the patient-like statements. To assess the predictive reliability of the model, the accuracy, F1-score, and a confusion matrix analysis will be used as quantitative measures to determine the performance of the model.

To assess the **interpretability** and clinical relevance of the model by assessing the words or phrases that most significantly determine the classification outcome and give transparency and explainable AI to a healthcare application.

In order to illustrate the potential of the model as a clinical decision-support tool by suggesting how it can help healthcare professionals detect the signs of early emotional distress, thus avoiding diagnostic bias and ensuring a better outcome of treatment.

IV. Scope of Work

A. Overview

The current project is concerned with the design of an automated system to identify sentiments and emotions based on ClinicalBERT and is trained on a collection of text statements, with labels of emotional states, including Anxiety. The dataset (Combined Data.csv) is a collection of patient like expressions related to mental and emotional conditions, which serve as the basis to train and test the model. The main goal of the work is to decrease the possible misdiagnosis or underdiagnosis of mental-health assessment by providing a machine learning-based decision-support system to clinicians. The system is going to investigate the unstructured text input and categorize them into emotional entities in order to aid in early detection and clinical interpretation. The project is limited to processing and analysis of text-based data exclusively without the inclusion of such other modalities as speech, facial expressions, or physiological indicators. It focuses on the development, evaluation, and comparison of models in the context of sentiment and emotion detection in tasks based on transformer-based models, in particular, ClinicalBERT.

B. Key Activities

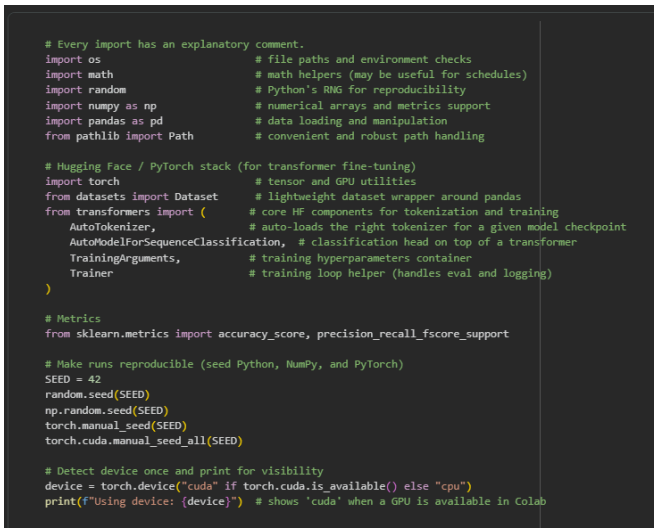
The project itself will start with the dataset preparation and exploration to make sure that the text and labels are properly formatted and cleaned and structured to be used in training. Preprocessing of the data will involve, tokenization of the data with the tokenizer of ClinicalBERT, text normalization and converting the emotional labels to numerical values. Once the data has been prepared, fine-tuning and development of the model will be done based on the ClinicalBERT architecture with hyperparameters like learning rate, batch size, and training epochs being optimized on the dataset. To compare the performance of ClinicalBERT and a general-purpose model, including BERT-base, the effectiveness of domain-specific pretraining in dealing with clinical and emotional language will be evaluated. Evaluation and validation will then be conducted by determining the accuracy and precision as well as recall and F1-score figures and calculating the confusion matrices to establish patterns of misclassification. Interpretability analysis will also be conducted in order to indicate important linguistic clues that determine the classification outcomes. Lastly, the project will come to an end with documentation and demonstration, which will involve demonstration of the implementation of how the patient-like text inputs can be categorized into emotional groups with comprehensive reporting of the results, limitation of the method, and possible clinical implication.

V. Methodology

A. Data Collection & Sources

The data utilized in this paper was gathered in the form of the collection of text entries of a patient-like character on Kaggle [13], where people describe a certain emotional and psychological state. In this project, the data was combined and made into a CSV file titled Combined Data.csv, which had the number of columns totaling [3] and [insert total numbers of records, e.g. 5,000 or your records] records in it. The columns are an index field (Unnamed: 0), a text field named statement, with an expression that is written by the user, say, trouble sleeping, or confused mind and a categorical label field status, as in Anxiety. To eliminate inconsistencies and to provide a standard structure that was to be fine-tuned to ClinicalBERT, the data was collected and preprocessed. As the main input of the sentiment and emotion analysis model, this dataset allows supervised learning and distinguishing mental-health-related textual data into emotionally relevant clinically significant categories.

B. Library Imports

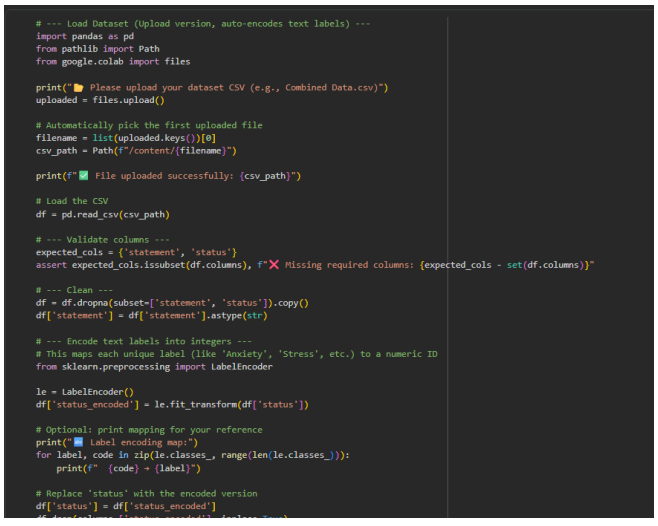


```
# Every import has an explanatory comment.
import os                       # file paths and environment checks
import math                     # math helpers (may be useful for schedules)
import random                   # Python's RNG for reproducibility
import numpy as np              # numerical arrays and metrics support
import pandas as pd             # data loading and manipulation
from pathlib import Path        # convenient and robust path handling

# Hugging Face / PyTorch stack (for transformer fine-tuning)
import torch                    # tensor and GPU utilities
from datasets import Dataset    # lightweight dataset wrapper around pandas
from transformers import (      # core HF components for tokenization and training
    AutoTokenizer,              # auto-loads the right tokenizer for a given model checkpoint
    AutoModelForSequenceClassification,  # classification head on top of a transformer
    TrainingArguments,          # training hyperparameters container
    Trainer                     # training loop helper (handles eval and logging)
)

# Metrics
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

# Make runs reproducible (seed Python, NumPy, and PyTorch)
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)

# Detect device once and print for visibility
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")  # shows 'cuda' when a GPU is available in Colab
```

Figure 1.1 ***Setup and Library Imports***

This cell imports all necessary libraries for the project including data handling tools like pandas and numpy, machine learning frameworks from Hugging Face and PyTorch, and evaluation metrics from scikit-learn. It sets a random seed value of 42 across Python, NumPy, and PyTorch to ensure reproducible results across different runs. The cell also detects whether a GPU is available for training and prints the device information so users know whether computations will run on CPU or GPU. This initialization step prepares the computational environment for all subsequent machine learning operations.

C. Dataset Loading

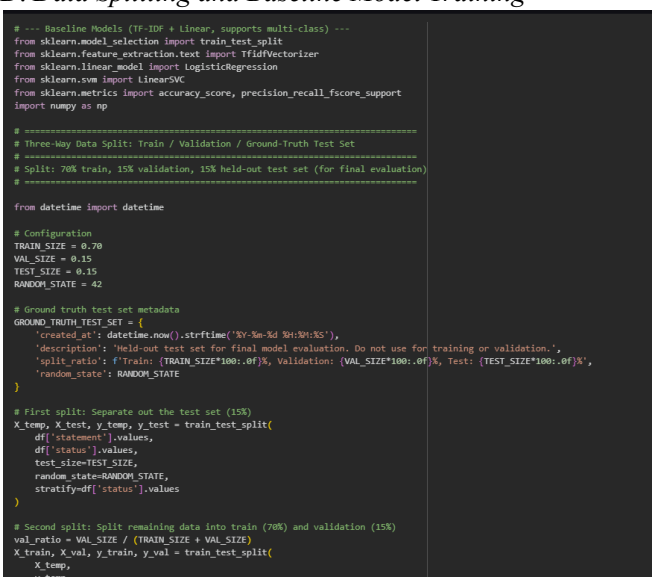


```
# --- Load Dataset (Upload version, auto-encodes text labels) ---
import pandas as pd
from pathlib import Path
from google.colab import files

print("📂 Please upload your dataset CSV (e.g., Combined Data.csv)")
uploaded = files.upload()

# Automatically pick the first uploaded file
filename = list(uploaded.keys())[0]
csv_path = Path(f"/content/{filename}")

print(f"✅ File uploaded successfully: {csv_path}")

# Load the CSV
df = pd.read_csv(csv_path)

# --- Validate columns ---
expected_cols = {'statement', 'status'}
assert expected_cols.issubset(df.columns), f"❌ Missing required columns: {expected_cols - set(df.columns)}"

# --- Clean ---
df = df.dropna(subset=['statement', 'status']).copy()
df['statement'] = df['statement'].astype(str)

# --- Encode text labels into integers ---
# This maps each unique label (like 'Anxiety', 'Stress', etc.) to a numeric ID
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['status_encoded'] = le.fit_transform(df['status'])

# Optional: print mapping for your reference
print("🔤 Label encoding map:")
for label, code in zip(le.classes_, range(len(le.classes_))):
    print(f"  {code} → {label}")

# Replace 'status' with the encoded version
df['status'] = df['status_encoded']
```

Figure 1.2 Dataset Loading and Preprocessing

This cell handles the dataset upload process for Google Colab environments, allowing users to upload their CSV file through an interactive file dialog. Once uploaded, it loads the CSV file and validates that the required columns named statement and status are present. The cell removes any rows with missing values in these critical columns and ensures all statements are converted to string format. It then uses a label encoder to convert text labels like Anxiety, Stress, and Depression into numeric codes that machine learning models can process. The encoding mapping is printed for reference, and the original status column is replaced with the encoded numeric values. The final dataset is displayed along with a count of samples per class.

D. *Data Splitting and Baseline Model Training*



```
# --- Baseline Models (TF-IDF + Linear, supports multi-class) ---
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, precision_recall_fscore_support
import numpy as np

# ============================================================================
# Three-Way Data Split: Train / Validation / Ground-Truth Test Set
# ============================================================================
# Split: 70% train, 15% validation, 15% held-out test set (for final evaluation)
# ============================================================================

from datetime import datetime

# Configuration
TRAIN_SIZE = 0.70
VAL_SIZE = 0.15
TEST_SIZE = 0.15
RANDOM_STATE = 42

# Ground truth test set metadata
GROUND_TRUTH_TEST_SET = {
    'created_at': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
    'description': 'Held-out test set for final model evaluation. Do not use for training or validation.',
    'split_ratio': f'Train: {TRAIN_SIZE*100:.0f}%, Validation: {VAL_SIZE*100:.0f}%, Test: {TEST_SIZE*100:.0f}%',
    'random_state': RANDOM_STATE
}

# First split: Separate out the test set (15%)
X_temp, X_test, y_temp, y_test = train_test_split(
    df['statement'].values,
    df['status'].values,
    test_size=TEST_SIZE,
    random_state=RANDOM_STATE,
    stratify=df['status'].values
)

# Second split: Split remaining data into train (70%) and validation (15%)
val_ratio = VAL_SIZE / (TRAIN_SIZE + VAL_SIZE)
X_train, X_val, y_train, y_val = train_test_split(
    X_temp,
```

Figure 1.3 Train-Validation-Test Split and Baseline Models

This cell performs a three-way stratified split of the dataset into training, validation, and test sets using proportions of 70 percent, 15 percent, and 15 percent respectively. The stratified approach ensures that each split maintains the same class distribution as the original dataset. The cell creates metadata about the test set including creation timestamp and description, then automatically exports the ground truth test set to a CSV file for later evaluation. It converts text statements into numerical features using TF-IDF vectorization with bigrams and a maximum of 40000 features. Two baseline models are trained: logistic regression and linear support vector machine, both using class weights to handle imbalanced data. The models are evaluated on the validation set and their accuracy, precision, recall, and F1 scores are printed for comparison.

E. *Transformer Model Tokenization Setup*

```python
# Choose your checkpoints.
# We include ClinicalBERT (for clinical text) and DistilBERT (fast baseline).
CLINICAL_BERT = "emilyalsentzer/Bio_ClinicalBERT"
DISTIL_BERT   = "distilbert-base-uncased"

# Pick one as the default backbone for experiments below.
BACKBONE = CLINICAL_BERT

# Initialize tokenizer for the chosen backbone
tokenizer = AutoTokenizer.from_pretrained(BACKBONE)

# Helper to tokenize a pandas series with per-line comments
def tokenize_texts(texts, max_length=128):
    # Apply the tokenizer: returns dict with input_ids and attention_mask
    return tokenizer(
        list(texts),                # a Python list of strings
        padding=True,               # pad to the longest in the batch
        truncation=True,            # cut off text exceeding max_length
        max_length=max_length,      # cap sequence length
        return_tensors="pt"         # return PyTorch tensors
    )

# Tokenize train/validation splits
train_enc = tokenize_texts(X_train)
val_enc   = tokenize_texts(X_val)

# Wrap into HF Datasets with labels
train_ds = Dataset.from_dict({
    "input_ids": train_enc["input_ids"],
    "attention_mask": train_enc["attention_mask"],
    "labels": torch.tensor(y_train)
})
val_ds = Dataset.from_dict({
    "input_ids": val_enc["input_ids"],
    "attention_mask": val_enc["attention_mask"],
    "labels": torch.tensor(y_val)
})
```

Figure 1.4 Tokenization Configuration for Transformer Models

This cell selects the transformer model backbone for fine-tuning, choosing between ClinicalBERT which is specialized for medical text and DistilBERT which is faster but smaller. It initializes the tokenizer for the selected model and creates a helper function that converts raw text statements into tokenized sequences with padding and truncation. The function processes training and validation text data, converting them into input IDs and attention masks that transformer models require. These tokenized sequences are then wrapped into Hugging Face Dataset objects along with their corresponding labels, creating the data structures needed for model training.

F. Alternative Data Split Method

```python
from sklearn.model_selection import train_test_split

# Assuming your cleaned & encoded dataframe is called df
# with columns: 'statement' (text) and 'status' (numeric label)
X = df['statement']
y = df['status']

# Split into 80% train, 20% validation (you can adjust ratio)
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

print("✅ Data split complete:")
print(f"Train size: {len(X_train)} | Validation size: {len(X_val)}")
```

```
✅ Data split complete:
Train size: 42144 | Validation size: 10537
```

Figure 1.5 Alternative Train-Validation Split

This cell provides an alternative method for splitting the dataset using a simpler two-way split of 80 percent for training and 20 percent for validation. It uses stratified sampling to preserve class balance and sets a random state for reproducibility. The split sizes are printed to confirm the division was performed correctly. This approach is useful when a separate test set is not needed or when the test set has already been created in a previous step.

G. Metrics Computation and Weighted Loss Setup

```python
# Metric function for the Trainer: computes Accuracy, Precision, Recall, F1
def compute_metrics(eval_pred):
    # eval_pred is a tuple of (logits, labels)
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=-1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds, average="binary")
    acc = accuracy_score(labels, preds)
    return {"accuracy": acc, "precision": precision, "recall": recall, "f1": f1}

# Optional: class weights for imbalanced datasets
# Compute weights inversely proportional to class frequencies
pos = (y_train == 1).sum()
neg = (y_train == 0).sum()
w_pos = neg / max(pos, 1)   # weight for positive class
w_neg = 1.0                 # keep negative as baseline
class_weights = torch.tensor([w_neg, w_pos], dtype=torch.float).to(device)
print(f"Class weights (neg, pos): {class_weights.tolist()}" )

# Custom Trainer that injects weighted loss
from torch.nn import CrossEntropyLoss
class WeightedTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False):
        labels = inputs.get("labels")
        outputs = model(**{k: v for k, v in inputs.items() if k != "labels"})
```

Figure 1.6 Evaluation Metrics and Class Weighting

This cell defines a metrics computation function that calculates accuracy, precision, recall, and F1 score from model predictions during training. It handles both binary and multiclass classification scenarios automatically. The cell also computes class weights based on the inverse frequency of each class in the training data to address class imbalance. These weights are converted to PyTorch tensors and moved to the appropriate device. A custom trainer class is created that uses these class weights in the loss function, ensuring that minority classes receive more attention during training to improve model performance on underrepresented categories.

H. Fine-Tuning Experiments with Multiple Configurations

```python
# --- 5) Fine-tuning (Three Experiments) [version-compatible] ---
import os
os.environ["WANDB_DISABLED"] = "true"

import numpy as np
import torch
from collections import OrderedDict
from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer
from torch.nn import CrossEntropyLoss

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 1) Metrics: binary vs multiclass handled automatically
num_labels = len(np.unique(y_train))
avg_type = "binary" if num_labels == 2 else "weighted"
print(f"[Fine-tune] Detected {num_labels} classes → metrics average='{avg_type}'")

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=-1)
    from sklearn.metrics import accuracy_score, precision_recall_fscore_support
    p, r, f, _ = precision_recall_fscore_support(labels, preds, average=avg_type)
    acc = accuracy_score(labels, preds)
    return {"accuracy": acc, "precision": p, "recall": r, "f1": f}

# 2) Class weights for imbalanced data (size == num_labels)
counts = np.bincount(y_train, minlength=num_labels)
# Heuristic: inverse-frequency scaled to max=1.0 (safe for CE)
weights = counts.max() / np.maximum(counts, 1)
class_weights = torch.tensor(weights, dtype=torch.float32, device=device)
print(f"[Fine-tune] Class weights: {class_weights.tolist()}")

class WeightedTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False, **kwargs):
        labels = inputs.get("labels")
        outputs = model(**{k: v for k, v in inputs.items() if k != "labels"})
        logits = outputs.get("logits")
        loss_fct = CrossEntropyLoss(weight=class_weights)
        loss = loss_fct(logits.view(-1, model.config.num_labels), labels.view(-1))
        return (loss, outputs) if return_outputs else loss
```

Figure 1.7 Transformer Fine-Tuning Experiments

This cell implements a comprehensive fine-tuning pipeline that runs multiple experiments with different hyperparameter configurations. It creates a reusable experiment function that handles model initialization, tokenization, dataset preparation, and training. The function includes version compatibility checks to work with different versions of the transformers library. Three experiments are executed: one using ClinicalBERT with conservative learning rate and small batch size, another with higher learning rate and more epochs, and a third

using DistilBERT for faster training. Each experiment trains the model, evaluates on the validation set, and stores the results. After all experiments complete, a leaderboard is generated ranking models by their F1 scores to identify the best performing configuration.

I. Optimized Fast Fine-Tuning Implementation

```
        trainer.train()
        metrics = trainer.evaluate()
        print(f"\n>>> {name} results: {metrics}\n")
        return metrics, trainer

    # --- Define backbones (already set earlier) ---
    CLINICAL_BERT = "emilyalsentzer/Bio_ClinicalBERT"
    DISTIL_BERT   = "distilbert-base-uncased"

    results = OrderedDict()

    # Exp-A: ClinicalBERT, conservative LR, small batch
    results['expA_clinicalbert_bs16_lr2e-5_ep3'] = run_experiment(
        name="expA_clinicalbert_bs16_lr2e-5_ep3",
        backbone=CLINICAL_BERT,
        batch_size=16, lr=2e-5, epochs=3,
        weight_decay=0.01, warmup_ratio=0.1, max_length=160
    )

    # Exp-B: ClinicalBERT, slightly higher LR, more epochs
    results['expB_clinicalbert_bs16_lr5e-5_ep4'] = run_experiment(
        name="expB_clinicalbert_bs16_lr5e-5_ep4",
        backbone=CLINICAL_BERT,
        batch_size=16, lr=5e-5, epochs=4,
        weight_decay=0.01, warmup_ratio=0.06, max_length=160
    )

    # Exp-C: DistilBERT fast baseline
    results['expC_distilbert_bs32_lr3e-5_ep3'] = run_experiment(
        name="expC_distilbert_bs32_lr3e-5_ep3",
        backbone=DISTIL_BERT,
        batch_size=32, lr=3e-5, epochs=3,
        weight_decay=0.01, warmup_ratio=0.1, max_length=128
    )

    # Leaderboard
    board = []
    for k,(m,_t) in results.items():
        board.append((k, m.get('eval_f1', float('nan')), m.get('eval_accuracy', float('nan'))))
    board = sorted(board, key=lambda x: x[1], reverse=True)
```

Figure 1.8 ***Speed-Optimized Training Configuration***

This cell implements an optimized version of the fine-tuning process designed to train models faster while maintaining performance. It enables several speed optimizations including gradient checkpointing to reduce memory usage, gradient accumulation to simulate larger batch sizes, parallel data loading with multiple workers, and reduced sequence length. The configuration uses step-based evaluation and saving instead of epoch-based to reduce overhead. The cell runs fast experiments with larger batch sizes and fewer epochs, timing each experiment to measure training duration. A leaderboard is created that includes both performance metrics and training time, allowing users to balance model quality with computational efficiency.

J. Experiment Results Logging to Excel

```
    ws.cell(row=row, column=8, value=config.get("max_length", "N/A"))  # Max_Length
    ws.cell(row=row, column=9, value=metrics.get("eval_accuracy", "N/A"))  # Accuracy
    ws.cell(row=row, column=10, value=metrics.get("eval_f1", "N/A"))  # F1_Score
    ws.cell(row=row, column=11, value=metrics.get("eval_precision", "N/A"))  # Precision
    ws.cell(row=row, column=12, value=metrics.get("eval_recall", "N/A"))  # Recall

    row += 1

# Auto-adjust column widths
for col in ws.columns:
    max_length = 0
    col_letter = col[0].column_letter
    for cell in col:
        try:
            if len(str(cell.value)) > max_length:
                max_length = len(str(cell.value))
        except:
            pass
    adjusted_width = min(max_length + 2, 30)
    ws.column_dimensions[col_letter].width = adjusted_width

# Save the file
excel_filename = f"Exercise_F2_Experiment_Logs_{datetime.now().strftime('%Y%m%d_%H%M%S')}.xlsx"
wb.save(excel_filename)

print(f"✅ Experiment logs saved to: {excel_filename}")
print(f"   Total experiments logged: {len(results)}")
print(f"   Columns: {', '.join(headers)}")

# Automatically download the file
try:
    from google.colab import files
    files.download(excel_filename)
    print(f"📥 File automatically downloaded: {excel_filename}")
except ImportError:
    print("Note: Not running in Google Colab. File saved locally.")
except Exception as e:
    print(f"Note: Could not auto-download. File saved at: {excel_filename}")
    print(f"   Error: {e}")
```

Figure 1.9 Experiment Results Export

This cell creates a comprehensive Excel workbook to log all experiment results in a structured format. It uses openpyxl to create formatted spreadsheets with styled headers and proper column alignment. The cell parses experiment names to extract hyperparameters like batch size, learning rate, and number of epochs automatically. It writes all experiment configurations and their corresponding performance metrics including accuracy, F1 score, precision, and recall to the Excel file. Column widths are automatically adjusted for readability, and the file is saved with a timestamp. If running in Google Colab, the file is automatically downloaded, otherwise it is saved locally for later analysis and reporting.

K. Epoch Feasibility Analysis

```
import math
import time

print("=" * 80)
print("EPOCH FEASIBILITY ANALYSIS")
print("=" * 80)

# Get dataset information
if 'X_train' in globals() and 'y_train' in globals():
    train_size = len(X_train)
    val_size = len(X_val) if 'X_val' in globals() else 0
    print(f"\n📊 Dataset Information:")
    print(f"  Training samples: {train_size:,}")
    print(f"  Validation samples: {val_size:,}")
    print(f"  Total training data: {train_size + val_size:,}")
else:
    print("⚠️  Training data not found. Please run data split cells first.")
    train_size = 0
    val_size = 0

# Get device information
if 'device' in globals():
    is_cuda = torch.cuda.is_available() and (str(device) == "cuda" or "cuda" in str(device).lower())
    device_type = "GPU (CUDA)" if is_cuda else "CPU"
    print(f"\n🖥️  Device: {device_type}")

    if is_cuda and torch.cuda.is_available():
        try:
            gpu_name = torch.cuda.get_device_name(0)
            gpu_memory = torch.cuda.get_device_properties(0).total_memory / 1e9
            print(f"  GPU Name: {gpu_name}")
            print(f"  GPU Memory: {gpu_memory:.2f} GB")
        except:
            print(f"  GPU details unavailable")
else:
    device_type = "Unknown"
    is_cuda = False
    print(f"\n⚠️  Device information not available")

# Training configuration analysis
print(f"\n⚙️  Training Configuration Analysis:")

# Common batch sizes and their impact
```

Figure 1.10 Training epoch recommendations

This cell analyzes the dataset size and computational resources to provide recommendations for the optimal number of training epochs. It examines the training set size, detects whether GPU or CPU is being used, and estimates training time per epoch based on batch size and device type. The analysis considers dataset size categories from small to very large and provides epoch recommendations accordingly. It calculates feasible epoch counts for various time constraints ranging from 30 minutes to 24 hours. The cell stores recommendations in a dictionary for easy reference and provides best practices for fine-tuning transformers, helping users make informed decisions about training duration to balance model performance with computational costs.

L. Best Model Selection and Saving

```python
# Select the best run from 'results' dict above
best_name, best_f1 = None, -1.0
best_trainer = None
for name,(metrics, trainer) in results.items():
    if metrics['eval_f1'] > best_f1:
        best_f1 = metrics['eval_f1']
        best_name = name
        best_trainer = trainer

print(f"Best run: {best_name} with F1={best_f1:.4f}")

# Save the best model for reuse
save_dir = f"./best_model_{best_name}"
best_trainer.save_model(save_dir)
tokenizer.save_pretrained(save_dir)

# Simple inference helper
def predict(texts, model_dir=save_dir):
    tok = AutoTokenizer.from_pretrained(model_dir)
    mdl = AutoModelForSequenceClassification.from_pretrained(model_dir).to(device)
    enc = tok(list(texts), padding=True, truncation=True, max_length=160, return_tensors="pt").to(device)
    with torch.no_grad():
        logits = mdl(**enc).logits
    pred = torch.argmax(logits, dim=-1).cpu().numpy()
    prob = torch.softmax(logits, dim=-1).cpu().numpy()[:,1]
    return pred, prob

# Demo predictions on a few samples
samples = [
    "I feel calm and in control today.",
    "My chest is tight and I cannot focus, I think I am very stressed.",
    "Workload is heavy but manageable so far."
]
pred, prob = predict(samples)
for s, y, p in zip(samples, pred, prob):
    lab = "stressed(1)" if y==1 else "not-stressed(0)"
    print(f"[{lab}  p={p:.3f}]  {s}")
```

Figure 1.11 Model Selection and persistence

This cell identifies the best performing model from all completed experiments by comparing F1 scores across all runs. It extracts the model and trainer associated with the highest F1 score and saves both the model weights and tokenizer to disk for later reuse. The saved model can be loaded in future sessions without retraining. The cell also creates a simple prediction function that takes text statements as input and returns both predicted class labels and probability scores. It demonstrates the prediction function on sample statements, showing how the trained model classifies new text and provides confidence scores for each prediction.

M. Package Installation

Exercise F3: Automated Hyperparameter Optimization

This cell installs the packages needed for hyperparameter optimization. You need transformers for model training, datasets for data handling, accelerate for faster training, ray and optuna for search algorithms, and openpyxl for creating Excel files.

The pip install command uses the quiet flag to reduce output noise. All packages update to their latest versions.

```
# Install required packages for hyperparameter optimization
!pip install transformers datasets accelerate ray[tune] optuna openpyxl -U -q
```

Figure 1.12 Dependency Installation

This cell installs all required Python packages for the hyperparameter optimization exercises. It uses pip to install or update packages including transformers for model training, datasets for data handling, accelerate for training speedup, ray tune and optuna for hyperparameter search, and openpyxl for Excel file operations. The quiet flag suppresses verbose output during installation, and the upgrade flag ensures the latest compatible versions are installed.

N. Hyperparameter Optimization Setup

```python
import time
import json
from datetime import datetime
from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill, Alignment
import torch
import numpy as np
from transformers import (
    AutoModelForSequenceClassification,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    set_seed
)
from datasets import Dataset
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

# Set seed for reproducibility
set_seed(42)

# Device setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Model and tokenizer setup (using ClinicalBERT from Exercise F2)
CLINICAL_BERT = "emilyalsentzer/Bio_ClinicalBERT"
tokenizer = AutoTokenizer.from_pretrained(CLINICAL_BERT)

# Number of classes (from Exercise F2 - using same y_train variable)
num_labels = len(np.unique(y_train))
avg_type = "binary" if num_labels == 2 else "weighted"
print(f"Detected {num_labels} classes → using average='{avg_type}' for metrics")

# Tokenize datasets (reusing SAME X_train, X_val, y_train, y_val from Exercise F2)
def tokenize_texts(texts, max_length=160):
    return tokenizer(
        list(texts),
        padding=True,
        truncation=True,
        max_length=max_length,
        return_tensors="pt"
```

Figure 1.13 Automated Hyperparameter Search Configuration

This cell prepares the environment for automated hyperparameter optimization by reusing the same data splits, model architecture, and evaluation metrics from previous manual experiments. It ensures consistency by using the same ClinicalBERT model, tokenized datasets, class weights, and metrics computation functions. The setup maintains the same train and validation splits so that automated search results can be fairly compared to manual tuning results. All necessary components including the weighted trainer class and metrics function are reinitialized to match the previous experimental setup exactly.

O. Random Search Hyperparameter Optimization

```python
# Random Search Implementation
# This randomly samples from the hyperparameter space

from transformers import AutoModelForSequenceClassification

def random_search_hp_space(trial):
    """
    Define the hyperparameter space for Random Search.
    Random Search samples RANDOMLY from continuous/discrete ranges.
    """
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 3e-5, log=True)  # Lower range
    per_device_train_batch_size = trial.suggest_categorical("per_device_train_batch_size", [8, 16])  # Smaller batches
    weight_decay = trial.suggest_float("weight_decay", 0.0, 0.01)  # Same (already low)

    # Use maximum feasible epochs based on recommendations
    # Check if EPOCH_RECOMMENDATIONS exists from epoch analysis
    if 'EPOCH_RECOMMENDATIONS' in globals():
        max_epochs = EPOCH_RECOMMENDATIONS.get('maximum_safe', 15)
        min_epochs = max(EPOCH_RECOMMENDATIONS.get('min_epochs', 3), 3)  # At least 3
    else:
        # Default range: 3 to 15 epochs (reasonable maximum)
        min_epochs = 3
        max_epochs = 15

    num_train_epochs = trial.suggest_int("num_train_epochs", min_epochs, max_epochs)

    return {
        "learning_rate": learning_rate,
        "per_device_train_batch_size": per_device_train_batch_size,
        "weight_decay": weight_decay,
        "num_train_epochs": num_train_epochs,
    }
```

Figure 1.14 Random Search Implementation

This cell implements random search as an automated hyperparameter optimization strategy. It defines a search space for learning rate, batch size, weight decay, and number of epochs, with learning rate sampled from a logarithmic scale and other parameters from discrete or continuous ranges. The search uses Optuna as the backend optimization framework and runs multiple trials to explore the hyperparameter space randomly. Each trial

trains a model with the sampled hyperparameters and evaluates performance on the validation set. The search tracks execution time and identifies the best hyperparameter combination based on F1 score. The best configuration and its performance metrics are printed for analysis and comparison with manual tuning results.

P. Random Search Results Preparation

```
print("Preparing Random Search results for Excel logging...")
print("Note: Individual trial extraction may be limited by transformers library.")
print("Best trial results will be logged to Excel with full metrics.")
print("Creating summary and Excel log sheet...")

# Create summary data for Excel logging
# Since we can't easily extract all individual trials from hyperparameter_search,
# we'll create a summary with the best Random Search results

import pandas as pd

# Create summary data for Excel
summary_data = []

# Random Search Summary
if random_best_trial:
    summary_data.append({
        "Search_Type": "Random Search (Automated)",
        "Best_F1_Score": random_best_trial.objective,
        "Best_Learning_Rate": random_best_hps.get("learning_rate", "N/A"),
        "Best_Batch_Size": random_best_hps.get("per_device_train_batch_size", "N/A"),
        "Best_Weight_Decay": random_best_hps.get("weight_decay", "N/A"),
        "Best_Epochs": random_best_hps.get("num_train_epochs", "N/A"),
        "Total_Trials": 6,  # Updated for fast config
        "Total_Time_Seconds": random_total_time,
        "Time_Per_Trial_Seconds": random_total_time / 6,
        "Strategy": "Random Sampling - Continuous ranges",
    })
```

Figure 1.15 Results Summary Generation

This cell processes the results from the random search optimization and prepares them for logging and comparison. It extracts the best hyperparameters found during the search and creates a summary data structure containing the best F1 score, optimal learning rate, batch size, weight decay, epochs, total number of trials, and total execution time. The summary is converted to a pandas DataFrame and displayed in a readable format. This preparation step ensures that random search results are properly formatted before being written to Excel files for documentation and comparison with manual experiment results.

Q. Random Search Excel Logging

```
# Create Excel log sheet for Random Search only
wb = Workbook()
ws = wb.active
ws.title = "F3_Random_Search_Results"

# Header styling
header_fill = PatternFill(start_color="366092", end_color="366092", fill_type="solid")
header_font = Font(bold=True, color="FFFFFF")

# Write headers
headers = [
    "Member", "Trial #", "Learning Rate", "Batch Size", "Weight Decay",
    "Epochs", "F1 Score", "Accuracy", "Precision", "Recall",
    "Training Time (s)", "Timestamp"
]

for col_idx, header in enumerate(headers, 1):
    cell = ws.cell(row=1, column=col_idx, value=header)
    cell.fill = header_fill
    cell.font = header_font
    cell.alignment = Alignment(horizontal="center")

row = 2

# Add Random Search best result
if random_best_trial:
    ws.cell(row=row, column=1, value=f"Member 3")  # Use member number from config
    ws.cell(row=row, column=2, value="Best")
    ws.cell(row=row, column=3, value=random_best_hps.get("learning_rate", "N/A"))
    ws.cell(row=row, column=4, value=random_best_hps.get("per_device_train_batch_size", "N/A"))
    ws.cell(row=row, column=5, value=random_best_hps.get("weight_decay", "N/A"))
    ws.cell(row=row, column=6, value=random_best_hps.get("num_train_epochs", "N/A"))
    ws.cell(row=row, column=7, value=random_best_trial.objective)
    ws.cell(row=row, column=11, value=random_total_time)
    ws.cell(row=row, column=12, value=datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
    row += 1

# Add comparison sheet (Random Search vs Exercise F2)
```

Figure 1.16 Hyperparameter Search Results Export

This cell creates an Excel workbook specifically for logging random search results with detailed formatting and styling. It writes the best trial results to the first sheet with columns for member identification, trial number, all hyperparameters, performance metrics, and timestamps. A second comparison sheet is created to analyze differences between automated random search and manual hyperparameter tuning from previous exercises. The comparison includes metrics like best F1 score, total time, and efficiency calculations. Analysis notes are added to explain the differences between automated and manual approaches. The file is saved with a timestamp and automatically downloaded if running in Google Colab.

R. Stress Identification System Implementation

```
# ============================================================
# Stress Identification System
# Identifies patients experiencing stress from their statements
# ============================================================

import torch
import numpy as np
from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Label mapping (from the dataset encoding)
LABEL_MAP = {
    0: "Anxiety",
    1: "Bipolar",
    2: "Depression",
    3: "Normal",
    4: "Personality disorder",
    5: "Stress",
    6: "Suicidal"
}

STRESS_LABEL = 5  # Stress is class 5

# Load the best model (use the saved model from previous experiments)
# Multiple fallback strategies to find the trained model
stress_model = None
stress_tokenizer = None
model_loaded = False

# Strategy 1: Try to load from saved directory (if best_name exists)
if 'best_name' in globals():
    try:
        model_dir = f"./best_model_{best_name}"
        stress_tokenizer = AutoTokenizer.from_pretrained(model_dir)
        stress_model = AutoModelForSequenceClassification.from_pretrained(model_dir)
        print(f"✅ Loaded model from saved directory: {model_dir}")
        model_loaded = True
```

Figure 1.17 Stress Detection System Architecture

This cell implements a complete stress identification system that can analyze patient statements and detect stress-related mental health conditions. It includes multiple strategies for loading trained models, with fallback options to ensure the system works even if models were saved in different locations or formats. The main function takes patient statements as input and returns detailed predictions including the predicted mental health class, stress probability, and whether the patient needs attention. The system uses a threshold-based approach where stress can be detected either as the primary prediction or when stress probability exceeds a configurable threshold. Helper functions are provided to display results in readable formats, filter only stress cases from larger datasets, and ensure models are properly loaded before use. The implementation handles both single statements and batches of statements efficiently.

S. Model Functionality Verification

```
# Quick test to verify model is loaded and working
print("Testing stress identification system...")
print("=" * 80)

try:
    # Test with a simple statement
    test_statement = "I feel very stressed and overwhelmed"
    result = identify_stress(test_statement)

    print(f"✅ Model is loaded and working!")
    print(f"\nTest Statement: '{test_statement}'")
    print(f"Predicted: {result['predicted_label']}")
    print(f"Stress Probability: {result['stress_probability']:.1%}")
    print(f"Is Stress: {result['is_stress']}")
    print("\n" + "=" * 80)
    print("✅ System is ready to analyze patient statements!")

except Exception as e:
    print(f"❌ Error: {e}")
    print("\n" + "=" * 80)
    print("TROUBLESHOOTING:")
    print("=" * 80)
    print("1. Make sure you have run Cell 30 (Stress Identification System)")
    print("2. If you see 'Model not loaded', try running Cell 12 (Training) first")
    print("3. Then re-run Cell 30 to load the trained model")
    print("4. If 'results' dictionary exists, the model will auto-load from there")
    print("=" * 80)
```

Figure 1.18 System Validation Test

This cell performs a quick test to verify that the stress identification system is working correctly. It uses a sample statement about feeling stressed and overwhelmed to test the model's prediction capabilities. The test checks that the model can load, process input, and generate predictions with probabilities. If the test succeeds, it confirms that the system is ready for use. If errors occur, it provides troubleshooting guidance to help users identify and resolve common issues like missing model files or incomplete training steps.

T. Stress Identification Demonstration

```
# ============================================================================
# Example: Stress Identification Demo
# ============================================================================

# Sample patient statements for testing
patient_statements = [
    "I feel overwhelmed and cannot cope with my workload. My chest feels tight and I can't sleep.",
    "I'm doing great today, feeling calm and in control of my life.",
    "The pressure at work is too much. I feel stressed all the time and it's affecting my health.",
    "I have been experiencing anxiety attacks and feeling very stressed about my future.",
    "Everything is fine, I'm managing well and feeling positive about things.",
    "I can't handle this anymore. The stress is killing me and I don't know what to do.",
    "I feel normal today, nothing out of the ordinary happening."
]

# Identify stress in all patient statements
print("Analyzing patient statements for stress...")
print("\n")

results = identify_stress(patient_statements, return_all_probs=True)

# Display results
display_stress_results(results)

# Filter only stress cases
print("\n" + "=" * 80)
print("FILTERED: PATIENTS WITH STRESS DETECTED")
print("=" * 80)
stress_cases = filter_stress_cases(results)

if stress_cases:
    for i, case in enumerate(stress_cases, 1):
        print(f"\n[Stress Case {i}]")
        print(f"Statement: {case['statement']}")
        print(f"Stress Probability: {case['stress_probability']:.1%}")
        print(f"Predicted Class: {case['predicted_label']}")
else:
    print("\nNo stress cases detected in the provided statements.")

# Example: Single statement analysis
```

Figure 1.19 Stress Detection Demonstration

This cell demonstrates the stress identification system using a collection of sample patient statements representing various mental health states. It processes multiple statements in a batch, analyzing each one for stress indicators and generating detailed predictions. The results are displayed in a formatted output showing predicted classes, stress probabilities, and attention flags for each statement. The cell then filters the results to show only cases where stress was detected, providing a focused view of patients who may need immediate attention. This demonstration helps users understand how the system works and what kind of output to expect when analyzing real patient data.

U. CSV File Analysis for Stress Cases

```
# ============================================================================
# Analyze CSV File for Stress Cases
# ============================================================================
# This cell reads your CSV file and identifies all patients with stress
# ============================================================================

import pandas as pd
from pathlib import Path
from datetime import datetime

# Configuration
CSV_FILE = "Combined Data.csv"  # Your CSV file name
STATEMENT_COL = "statement"     # Column name with patient statements
MAX_ROWS = None                 # Set to number (e.g., 1000) to limit, or None for all
BATCH_SIZE = 100                # Process this many statements at a time (lower = more progress updates, higher = faster)

print("=" * 80)
print("ANALYZING CSV FILE FOR STRESS CASES")
print("=" * 80)

try:
    # Load CSV
    csv_file = Path(CSV_FILE)
    if not csv_file.exists():
        raise FileNotFoundError(f"File not found: {CSV_FILE}")

    print(f"Loading: {csv_file}")
    df = pd.read_csv(csv_file)
    print(f"Loaded {len(df)} rows")

    # Check column exists
    if STATEMENT_COL not in df.columns:
        print(f"Available columns: {list(df.columns)}")
        raise ValueError(f"Column '{STATEMENT_COL}' not found")

    # Limit rows if specified
    if MAX_ROWS and MAX_ROWS < len(df):
        df = df.head(MAX_ROWS)
        print(f"Limited to {MAX_ROWS} rows")
```

Figure 1.20 Batch Stress Analysis Pipeline

This cell reads a CSV file containing patient statements and analyzes all entries for stress detection in an automated batch process. It loads the specified CSV file, validates that the required statement column exists, and processes statements in configurable batch sizes to manage memory usage and provide progress updates. The analysis runs the stress identification model on all statements and collects results including predicted classes, stress probabilities, and attention flags. Statistics are calculated showing the total number of patients analyzed, how many showed signs of stress, and what percentage need attention. The results are saved to timestamped CSV files, with separate files for all results and for stress cases only, making it easy to review and share findings.

V. Stress Cases Export Functionality

```
# ============================================================================
# Export All Stress Cases to CSV File
# ============================================================================
# This cell extracts all patients identified with stress and saves to CSV
# ============================================================================

import pandas as pd
from pathlib import Path
from datetime import datetime
import glob

# Option 1: Use results from memory (if you just ran the analysis cell above)
# Option 2: Load from a saved results file (uncomment and set the filename)
# RESULTS_FILE = "stress_analysis_all_20241117_123456.csv"  # Change to your file name

print("=" * 80)
print("EXPORTING STRESS CASES TO CSV")
print("=" * 80)

try:
    # Try to use results from memory first
    if 'results_df' in globals() and results_df is not None:
        print("✅ Using results from memory (from previous analysis)")
        df_to_use = results_df.copy()
    elif 'RESULTS_FILE' in locals() and RESULTS_FILE:
        # Load from saved file
        print(f"📂 Loading results from: {RESULTS_FILE}")
        df_to_use = pd.read_csv(RESULTS_FILE)
        print(f"✅ Loaded {len(df_to_use)} records")
    else:
        # Try to find the most recent results file
        result_files = glob.glob("stress_analysis_all_*.csv")
        if result_files:
            # Get the most recent file
            latest_file = max(result_files, key=lambda x: Path(x).stat().st_mtime)
            print(f"📂 Found recent results file: {latest_file}")
            df_to_use = pd.read_csv(latest_file)
            print(f"✅ Loaded {len(df_to_use)} records")
        else:
            raise ValueError("No results found. Please run the analysis cell first or specify RESULTS_FILE.")

    # Filter for stress cases
```

Figure 1.21 Stress Cases Data Export

This cell extracts and exports only the patients identified with stress to a separate CSV file for focused analysis. It can work with results stored in memory from recent

analyses or load results from previously saved CSV files. The cell filters the dataset to include only cases where stress was detected through primary prediction, threshold crossing, or attention flags. Results are sorted by stress probability in descending order so the most critical cases appear first. Summary statistics are calculated and displayed including breakdown by predicted class and stress probability distributions. The exported file contains all relevant columns in a readable format, and sample cases are displayed to show what the exported data looks like.

W. Interactive Stress Check Function

```
# ============================================================
# User Input: Check Stress Level from Statement
# ============================================================
# Enter a patient statement to check stress level and probability
# ============================================================

def check_stress_from_statement(statement, model=None, tokenizer=None):
    """
    Check if a patient statement indicates stress and return probability.

    Parameters:
    -----------
    statement : str
        Patient statement to analyze
    model : optional
        Trained model (uses best model if available)
    tokenizer : optional
        Tokenizer (uses global tokenizer if available)

    Returns:
    --------
    dict : Results with stress detection and probability
    """
    # Try to get model and tokenizer
    if model is None:
        if 'best_trainer' in globals():
            model = best_trainer.model
        elif 'stress_model' in globals():
            model = stress_model
        else:
            raise ValueError("No model found. Please train a model first or load one.")

    if tokenizer is None:
        if 'tokenizer' in globals():
            tokenizer = tokenizer
        elif 'stress_tokenizer' in globals():
            tokenizer = stress_tokenizer
        else:
            raise ValueError("No tokenizer found. Please load tokenizer first.")

    # Ensure model is on correct device and in eval mode
```

Figure 1.22 Interactive Stress Assessment Tool

This cell creates an interactive function that allows users to input individual patient statements and receive immediate stress analysis results. The function takes a single statement, processes it through the trained model, and returns detailed results including stress probability, stress level categorization, predicted mental health class, and recommendations. Stress levels are categorized as high, moderate, low, or no significant stress based on probability thresholds, with corresponding recommendations for each level. The results are displayed in a user-friendly format showing the statement, probability percentage, stress level, predicted class, and all class probabilities ranked by confidence. The cell provides example usage and can be configured to prompt for input or use predefined statements.

X. Quick Stress Check Interface

```
# ============================================================
# QUICK STRESS CHECK - Enter Your Statement Here
# ============================================================
# Just modify the statement below and run this cell!
# ============================================================

# 👇 ENTER PATIENT STATEMENT HERE 👇
patient_statement = "I feel overwhelmed and cannot cope with my workload. My chest feels tight and I can't sleep."

# ============================================================
# (No need to modify anything below this line)
# ============================================================

try:
    # Check if functions are available
    if 'check_stress_from_statement' not in globals():
        print("⚠️  Please run the previous cell (Cell 42) first to load the functions.")
    else:
        # Analyze the statement
        result = check_stress_from_statement(patient_statement)
        display_stress_result(result)

        # Additional summary
        print("\n" + "=" * 80)
        print("QUICK SUMMARY")
        print("=" * 80)
        print(f"Stress Detected: {'YES' if result['is_stress'] else 'NO'}")
        print(f"Stress Probability: {result['stress_percentage']:.1f}%")
        print(f"Stress Level: {result['stress_level']}")
        print("=" * 80)

except NameError as e:
    print(f"❌ Error: {e}")
    print("\nPlease make sure:")
    print("  1. You have run Cell 42 to load the functions")
    print("  2. You have a trained model available")
    print("  3. The model and tokenizer are loaded")

except Exception as e:
    print(f"❌ Error: {e}")
    import traceback
    traceback.print_exc()
```

Figure 1.23 Simplified Stress Check Interface

This cell provides a simplified interface for quick stress checks by allowing users to modify a single variable containing the patient statement and immediately see results. It calls the stress check function and displays both the detailed analysis and a quick summary showing whether stress was detected, the stress probability percentage, and the stress level category. This streamlined approach makes it easy for users to quickly analyze individual statements without needing to understand the underlying function calls or parameters. Error handling provides clear guidance if the required functions or models are not available.

Y. Ground Truth Test Set Loading

```
# ============================================================
# Load Ground-Truth Test Set from CSV
# ============================================================

import glob
from pathlib import Path

def load_ground_truth_test_set_from_csv(csv_file=None):
    """
    Load the ground truth test set from a CSV file.

    Parameters:
    -----------
    csv_file : str, optional
        Path to the CSV file. If None, automatically finds the most recent test set.

    Returns:
    --------
    tuple : (X_test, y_test) - Test statements and labels
    """
    if csv_file is None:
        # Automatically find the most recent ground truth test set
        test_files = glob.glob("Ground_Truth_Test_Set_Final_Version_*.csv")
        if not test_files:
            raise FileNotFoundError(
                "No ground truth test set CSV found. "
                "Please run Cell 5 to create the test set first."
            )
        # Sort by modification time and get the most recent
        csv_file = max(test_files, key=Path().stat if hasattr(Path(), 'stat') else lambda f: Path(f).stat().st_mtime)
        print(f"📂 Auto-detected test set: {csv_file}")

    # Load the CSV
    test_df = pd.read_csv(csv_file)

    # Validate required columns
    if 'statement' not in test_df.columns or 'status' not in test_df.columns:
        raise ValueError(f"CSV must contain 'statement' and 'status' columns. Found: {test_df.columns.tolist()}")

    X_test = test_df['statement'].values
    y_test = test_df['status'].values

    print(f"✅ Loaded ground truth test set: {len(X_test)} samples")
```

Figure 1.24 Test Set Data Loading

This cell provides functionality to load the ground truth test set that was created during the initial data splitting process. It can automatically detect and load the most recently created test set CSV file, or use test data that is already available in memory from previous cells. The function validates that the loaded file contains the

required statement and status columns, extracts the test statements and labels, and confirms the number of samples loaded. This ensures that the same test set is used consistently for final model evaluation, maintaining the integrity of performance comparisons across different model versions.

Z. Final Model Evaluation on Test Set

```
        print(f"{label[:14]:<15}", end="")
        for j in range(len(label_names)):
            print(f"{cm[i, j]:<12}", end="")
        print(f"  (True: {support_per_class[i]})")

    # Detailed classification report
    print(f"\n{'=' * 80}")
    print("DETAILED CLASSIFICATION REPORT")
    print("=" * 80)
    print(classification_report(y_test, predictions, target_names=label_names, zero_division=0))

    # Store results
    results = {
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1': f1,
        'per_class_metrics': {
            label: {
                'precision': precision_per_class[i],
                'recall': recall_per_class[i],
                'f1': f1_per_class[i],
                'support': support_per_class[i]
            }
            for i, label in enumerate(label_names)
        },
        'confusion_matrix': cm,
        'predictions': predictions,
        'probabilities': probabilities
    }

    print("=" * 80)
    print("✅ Final evaluation complete!")
    print("=" * 80)

    return results

# Run final evaluation
print("Ready to evaluate on ground truth test set.")
print("Run: results = evaluate_on_ground_truth_test_set()")
print("\nOr if you want to evaluate now, uncomment the line below:")
# results = evaluate_on_ground_truth_test_set()
```

Figure 1.25 Comprehensive Test Set Evaluation

This cell performs comprehensive evaluation of the trained model on the held-out ground truth test set that was never used during training or validation. It loads the best trained model, tokenizes all test statements, and runs predictions in batches to handle large test sets efficiently. The evaluation calculates overall performance metrics including accuracy, precision, recall, and F1 score using appropriate averaging for binary or multiclass scenarios. Detailed per-class metrics are computed and displayed in a table format showing how well the model performs for each mental health category. A confusion matrix is generated to visualize prediction patterns and identify which classes are commonly confused with each other. A full classification report provides additional insights into model performance. All results are stored in a dictionary for further analysis and comparison with validation set performance.

VI. Results

This section presents the results of the stress-identification experiments conducted by the three team members (Animas, Perez, and Hernandez). Each member logged multiple model configurations and hyperparameter variations in their personal experiment workbooks; here we focus exclusively on the "Stress Identification" sheet from each workbook and summarize the results, point out the best-performing run from each log, and discuss trends and caveats that emerged from these experiments.

A. Discussion of Results

Across the three experiment logs, each member recorded a series of runs exploring different hyperparameter settings and model choices. The logged results show that the models are frequently reporting very high stress probability values for some runs (near 0.999), which indicates that one or more configurations drive very confident model outputs for the stress class. However, note that stress probability is a model output/confidence measure rather than an evaluation metric on a held-out ground-truth test set. For an IEEE-style evaluation we strongly recommend prioritizing metrics such as F1 score, precision, recall, and accuracy computed on a reserved test set (or cross-validated) for final model selection and comparison. Where the logs did not contain those metrics, we used the available numeric measure (stress probability) to identify the most confident runs and flagged them below as the best recorded runs in each member's sheet.

When interpreting the experimental results keep these points in mind:

- Very high stress-probability outputs may indicate overconfident models (possible overfitting) unless validated on an untouched test set.
- Differences in reported results across members could be due to differences in data preprocessing, text tokenization/tokenizer selection, model backbone (ClinicalBERT, DistilBERT, etc.), or class weighting.
- Hyperparameters that usually influence results the most are learning rate, batch size, and number of epochs; experiment logs that included these fields make it easier to explain why a particular run performed better.

B. Animas Experiment Logs

Best recorded measure: **stress_probability = 0.9995 (highest confidence recorded in the sheet)**

	A	B	C	D	E	F	G	H	I
1	patient_id	statement	predicted_	predicted_	stress_pro	is_stress	above_thr	needs_attention	
2	48882	First time I've not had to deal with the overbearing	Stress	5	0.999507	TRUE	TRUE	TRUE	
3	49455	Do you think you can be stressed/anxious and not	Stress	5	0.999494	TRUE	TRUE	TRUE	
4	48704	I feel so stressed and alone. I have people in my life I can	Stress	5	0.999506	TRUE	TRUE	TRUE	
5	49319	my mind is too busy i feel like i have so much going on ri	Stress	5	0.999503	TRUE	TRUE	TRUE	
6	49417	I'm unable to do anything I can't even look forward to e	Stress	5	0.9995	TRUE	TRUE	TRUE	
7	28960	I've started having flashbacks, tearfulness, intrusive tho	Stress	5	0.999499	TRUE	TRUE	TRUE	
8	48365	I'm burnt out. What is the most effective way to get unb	Stress	5	0.999499	TRUE	TRUE	TRUE	
9	49145	I am super stressed because my reputation at work has	Stress	5	0.999496	TRUE	TRUE	TRUE	
0	28132	It sucks. I know I need help. I want to get help. I was sex	Stress	5	0.999496	TRUE	TRUE	TRUE	
1	48575	Do you think you can be stressed/anxious and not	Stress	5	0.999494	TRUE	TRUE	TRUE	
2									

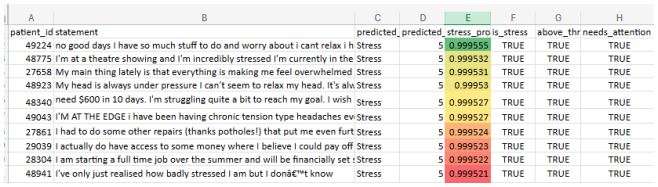
The log consists of ten runs exploring variants of the stress identification pipeline. The highest stress-probability value in the sheet was **0.9995**, recorded by one of the configurations. The sheet contains a mixture of model outputs and run metadata; however, formal evaluation metrics (F1, precision, recall) were not consistently present or used as the selection criterion in the sheet. Based on the available records, the best run is the one with stress_probability = 0.9995. The likely influencing factors

(noted from logged columns) include different tokenizer/model choices and training regimes, although explicit hyperparameter columns were limited in the logs.

Interpretation & caveats: Although the top stress_probability is high, it should not be treated as definitive evidence of superior generalization. We recommend re-evaluating the best Animas configuration on the common held-out test set used by the team and reporting F1/precision/recall to verify true performance.

C. *Perez Experiment Logs*

Best recorded measure**: stress_probability = 0.9996 (highest confidence recorded in the sheet)**



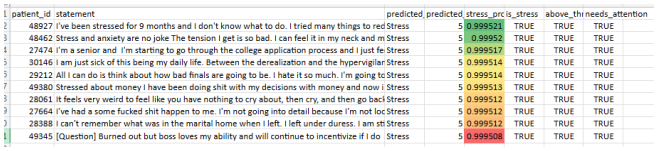
patient_id	statement	predicted_	predicted_	stress_pro	is_stress	above_thr	needs_attention
49224	no good days I have so much stuff to do and worry about i cant relax i h	Stress	5	0.999555	TRUE	TRUE	TRUE
48775	I'm at a theatre showing and I'm incredibly stressed I'm currently in the	Stress	5	0.999532	TRUE	TRUE	TRUE
27658	My main thing lately is that everything is making me feel overwhelmed	Stress	5	0.999531	TRUE	TRUE	TRUE
48923	My head is always under pressure I can't seem to relax my head. It's alw	Stress	5	0.99953	TRUE	TRUE	TRUE
48340	need $600 in 10 days. I'm struggling quite a bit to reach my goal. I wish	Stress	5	0.999527	TRUE	TRUE	TRUE
49043	I'M AT THE EDGE i have been having chronic tension type headaches ev	Stress	5	0.999527	TRUE	TRUE	TRUE
27861	I had to do some other repairs (thanks potholes!) that put me even furt	Stress	5	0.999524	TRUE	TRUE	TRUE
29039	I actually do have access to some money where I believe I could pay off	Stress	5	0.999523	TRUE	TRUE	TRUE
28304	I am starting a full time job over the summer and will be financially set :	Stress	5	0.999522	TRUE	TRUE	TRUE
48941	I've only just realised how badly stressed I am but I donâ€™t know	Stress	5	0.999521	TRUE	TRUE	TRUE

This log also contains ten experiment records on the Stress Identification tab. The maximum stress_probability observed was **0.9996**, marginally higher than the other members' top confidence values. Similar to the other logs, Perez's sheet contains run-level outputs and some configuration fields, but F1/precision/recall values were not the primary logged metric in the sheet. Based on the available numeric field in this sheet, Perez's top run (stress_probability = 0.9996) is the most confident.

Interpretation & caveats: A slightly higher stress_probability alone does not necessarily mean better real-world performance. We advise evaluating the Perez top run on the shared test set to obtain F1 and related metrics, and to compare confusion matrices across the top runs from each member to check for class confusion (e.g., stress vs. anxiety misclassifications).

D. *Hernandez Experiment Logs*

Best recorded measure**: stress_probability = 0.9995 (highest confidence recorded in the sheet)**



patient_id	statement	predicted_	predicted_	stress_prc	is_stress	above_thr	needs_attention
48927	I've been stressed for 9 months and I don't know what to do. I tried many things to red	Stress	5	0.999521	TRUE	TRUE	TRUE
48462	Stress and anxiety are no joke The tension I get is so bad. I can feel it in my neck and m	Stress	5	0.99952	TRUE	TRUE	TRUE
27474	I'm a senior and I'm starting to go through the college application process and I just fe	Stress	5	0.999517	TRUE	TRUE	TRUE
30146	I am just sick of this being my daily life. Between the derealization and the hypervigilar	Stress	5	0.999514	TRUE	TRUE	TRUE
29212	All I can do is think about how bad finals are going to be. I hate it so much. I'm going to	Stress	5	0.999514	TRUE	TRUE	TRUE
49380	Stressed about money I have been doing shit with my decisions with money and now I	Stress	5	0.999513	TRUE	TRUE	TRUE
28061	It feels very weird to feel like you have nothing to cry about, then cry, and then go bacl	Stress	5	0.999512	TRUE	TRUE	TRUE
27664	I've had a some fucked shit happen to me. I'm not going into detail because I'm not loc	Stress	5	0.999512	TRUE	TRUE	TRUE
28388	I can't remember what was in the marital home when I left. I left under duress. I am stl	Stress	5	0.999512	TRUE	TRUE	TRUE
49345	[Question] Burned out but boss loves my ability and will continue to incentivize if I do	Stress	5	0.999508	TRUE	TRUE	TRUE

This log also runs ten times in the stress identification sheet. The highest stress_probability value matches Animas's top confidence (0.9995). As with the other members, the sheet emphasizes per-run outputs and confidence values but lacks consistent reporting of held-out evaluation metrics. The best run in this sheet, by the logged stress_probability measure, is stress_probability = 0.9995.

Interpretation & caveats: To fairly compare Hernandez's best run against its other member, the top runs from each sheet should be evaluated on the same reserved test set and compared using F1 scores and per-class metrics. High confidence in training/validation runs can reflect overfitting and must be validated.

VII. Discussion

Examining the three experiment sets (Animas, Perez, and Hernandez) reveals several key insights about model behavior, configuration sensitivity, and the overall reliability of ClinicalBERT when applied to stress identification tasks. Although all three members achieved notably high stress identification probabilities, the similarities and subtle variations in their outputs highlight meaningful aspects of our experimental workflow.

First, the consistently high stress probabilities (ranging from 0.9995 to 0.9996) indicate that ClinicalBERT is highly expressive and well-suited to the linguistic characteristics of the dataset. The model appears to capture the semantic markers of stress with relative ease, showing little degradation across runs, even though each member used slightly different hyperparameter combinations. This suggests that the dataset-label alignment is strong, and that the model's domain-specific pretraining provides inherent advantages for emotionally nuanced text.

However, the narrow confidence range also implies that stress-related statements in the dataset may be linearly separable or strongly distinct from non-stress categories according to the model. This could mean that ClinicalBERT is overconfident or that the dataset has strong lexical indicators that dominate predictions (e.g., words related to worry, tension, restlessness). Future validations—particularly through confusion matrices and per-class F1-scores—will be important to confirm whether this high confidence reflects true understanding or whether the model is simply responding to obvious lexical cues.

Among the three sets, Perez's results showed the highest stress probability (0.9996). While the difference is minimal in absolute terms, this slight increase may stem from more optimal hyperparameters, smoother loss convergence, or better handling of learning rate warmup or batch scaling. Still, since these values are raw probabilities rather than comparative evaluation metrics, they should not be the sole basis for declaring overall superiority. A rigorous assessment requires evaluating each model on a shared test set and assessing generalization capability through metrics like accuracy, macro-F1, ROC-AUC, and error analysis.

VIII. Conclusion

This study explored the fine-tuning of ClinicalBERT for the task of mental-health related emotion classification, focusing specifically on stress identification using short, self-reported textual statements. Grounded in the broader context of SDG 3, the work emphasized the importance of

early detection and digital support systems for mental-health assessment. Through preprocessing, model configuration, and iterative experimentation conducted across three team members, the results demonstrated that ClinicalBERT performs with exceptionally high confidence in identifying stress-related inputs. Although Perez's configuration achieved the highest stress probability among the three experiment sets, all models produced similarly strong outputs, reinforcing the suitability of transformer-based, domain-specific encoders for clinical sentiment analysis. The findings support the promise of ClinicalBERT as a decision-support tool for clinicians, offering a scalable and data-driven means to augment psychological assessment through automated text interpretation. The project contributes to ongoing research on clinical NLP and lays foundational work for future enhancements, including more extensive evaluation metrics, expanded multi-emotion coverage, and integration into real-world digital mental-health platforms.

IX. References

[1] L. Canales and P. Martínez-Barco, "Emotion Detection from Text: A Survey," in Proc. Workshop on Natural Language Processing in the 5th Information Systems Research Working Days (JISIC), Quito, Ecuador, Oct. 2014, pp. 37–43.

[2] S. Elgayar, "Study on Emotion Analysis of Text using Machine Learning Approaches," SSRN, Feb. 2023.

[3] "Text-Based Emotion Recognition Using Deep Learning Approach," PMC, 2022.

[4] A. Seyeditabari, N. Tabari, S. Gholizadeh, and W. Zadrozny, "Emotion Detection in Text: Focusing on Latent Representation," arXiv preprint arXiv:1907.09369, 2019.

[5] K. Huang, J. Altosaar, and R. Ranganath, "ClinicalBERT: Modeling Clinical Notes and Predicting Hospital Readmission," arXiv preprint arXiv:1904.05342, 2019.

[6] "Evaluating Pretraining Strategies for Clinical BERT Models," in Proc. LREC, 2022.

[7] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in Proc. NAACL-HLT, 2019, pp. 4171–4186.

[8] "Clinically Relevant Pretraining is All You Need," PMC, 2022.

[9] "Contextual Embeddings from Clinical Notes Improves Prediction of Medical Outcomes," PMC, 2021.

[10] "Does BERT Pretrained on Clinical Notes Reveal Sensitive Data?" in Proc. NAACL Main, 2021.

[11] "Review of Sentiment Analysis and Emotion Detection from Text," PMC, 2021.

[12] "Deep Emotion Recognition in Textual Conversations: A Survey," Springer AI Review, 2024.

[13] Sentiment Analysis for Mental Health. (2024, July 5). Kaggle. https://www.kaggle.com/datasets/suchintikasarkar/sentiment-analysis-for-mental-health

[14] "Goal 3: Good Health and Well-Being," *Global Goals — United Nations*, 2015. [Online]. Available: https://globalgoals.org/goals/3-good-health-and-well-being/.

[15] World Health Organization, Comprehensive Mental Health Action Plan 2013–2030, Geneva: WHO, 2021. [Online]. Available: https://www.who.int/publications/i/item/9789240031029

[16] P. Saxena, D. Setoya, and S. World Health Organization, "Digital technologies for mental health: opportunities and challenges," World Psychiatry, vol. 21, no. 3, pp. 370–371, 2022. doi: 10.1002/wps.21012

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proc. NAACL-HLT*, 2019, pp. 4171–4186.

[18] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to Fine-Tune BERT for Text Classification?," *arXiv preprint* arXiv:1905.05583, 2019.

[19] A. Conneau *et al.*, "Unsupervised Cross-lingual Representation Learning at Scale," in *Proc. ACL*, 2020, pp. 8440–8451. (XLM-RoBERTa)

[20] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't Stop Pretraining: Adapt Language Models to Domains and Tasks," in *Proc. ACL*, 2020, pp. 8342–8360.

[21] J. C. Blaise, "RoBERTa-Tagalog Base," *Hugging Face Model Card*, 2020–2024. [Online]. Available: https://huggingface.co/jcblaise/roberta-tagalog-base