

# project-health-deterioration-model

June 12, 2025

## 1 0. Setup and Configuration

### Step 1: Install Core Dependencies

Install all required Python packages for AI-powered feature engineering, modeling, and NLP tasks.

```
[ ]: # Install Core Dependencies
!pip install -q numpy pandas matplotlib seaborn scikit-learn xgboost
↳ transformers imbalanced-learn tqdm
```

### Step 2: Set Working Directory

All generated data files (e.g., raw outputs, datasets, summaries) will be saved to this path for clarity and version control.

```
[ ]: # Set Working Directory in Colab/Drive
import os
my_file_path = "/content/drive/MyDrive/UM Data Science Course Information/
↳ WQD7005/Assignment Project/"
os.makedirs(my_file_path, exist_ok=True)
```

### Step 3: Authenticate Hugging Face

plan to access transformer models (e.g., MiniLM, BERT-tiny), login to Hugging Face Hub is required.

```
[ ]: # Setup Hugging Face Token
from huggingface_hub import notebook_login
notebook_login()
```

```
VBox(children=(HTML(value='<center> <img\>nsrc=https://huggingface.co/front/
↳ assets/huggingface_logo-noborder.svg...
```

### Step 4: Securely Load Azure API Credentials

Using secrets.json avoids hardcoding sensitive information. This supports secure API usage and easier sharing of my notebook.

```
[ ]: # Securely Load Azure API Credentials
# Azure endpoint and keys
import json
```

```

# Load secrets.json after upload
secrets_file = os.path.join(my_file_path, "secrets.json")
if os.path.exists(secrets_file):
    with open(secrets_file, "r") as f:
        secrets = json.load(f)

endpoint = secrets["AZURE_ENDPOINT"]
subscription_key = secrets["AZURE_KEY"]

```

### Step 5: Configure Azure OpenAI Client and Define GPT Prompt Wrapper

This step sets up the Azure OpenAI client and defines a reusable function for sending prompts to GPT-4o, enabling automated clinical text generation and interpretation.

```

[ ]: # Import Supporting Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta
from openai import AzureOpenAI
import random
import time

# Configure Azure OpenAI Client
api_version = "2024-12-01-preview"
deployment = "gpt-4o"
client = AzureOpenAI(
    api_version=api_version,
    azure_endpoint=endpoint,
    api_key=subscription_key,
)

# Prompt execution wrapper for reuse
def model_prompt(prompt, system_prompt="Act as a professional clinicians.",
    temperature=0.7, max_tokens=4096):
    response = client.chat.completions.create(
        model=deployment,
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": prompt}
        ],
        max_tokens=max_tokens,
        temperature=temperature,
    )
    return response.choices[0].message.content

```

### Step 6: Single Sample Data Generation via GPT (Validation Prompt)

To verify the response structure of GPT-4o by generating a realistic single-patient daily monitoring record, ensuring the output conforms to expected JSON schema for later batch generation.

```
[ ]: # Single Sample Data Generation via model
data_prompt = """
Generate a single, realistic patient monitoring record for one randomly
    ↪selected adult patient.

Provide the following fields:
- oxygen_saturation (in %)
- heart_rate (in bpm)
- temperature (in °C)
- blood_pressure (systolic/diastolic, e.g. "120/80")
- weight (in kg)
- blood_glucose (in mg/dL)

At the end, include a brief clinical_note (1-2 sentences, max 30 words)
    ↪summarizing the patient status based on the values above. Use professional
    ↪clinical tone with realistic variation (e.g. stable, recovering, mild
    ↪concerns).

Output as a valid JSON object with keys:
oxygen_saturation, heart_rate, temperature, blood_pressure, weight,
    ↪blood_glucose, clinical_note.

Constraints:
- Only output one JSON object.
- No markdown or explanation.
- Include realistic variation across different health conditions (e.g. fatigue,
    ↪post-op, dietary changes, stress).
- Ensure all fields are complete, no missing values.
"""

print(model_prompt(data_prompt))
```

```
{
  "oxygen_saturation": 94,
  "heart_rate": 88,
  "temperature": 37.5,
  "blood_pressure": "135/85",
  "weight": 76,
  "blood_glucose": 145,
  "clinical_note": "Patient exhibits mild tachycardia and elevated blood
glucose, likely linked to dietary factors or stress; overall vitals are stable
but require monitoring for potential metabolic concerns."
}
```

## 2 1. Dataset Simulation and Feature Engineering

Since a synthetic patient dataset with labeled `note_status` was already generated and preprocessed in the previous assignment, so this section focuses on loading the prepared dataset, checking basic data structure, verifying label quality, and ensuring readiness for AI-driven feature engineering and modeling.

### Step 1: Load Preprocessed Dataset

Load the previously prepared patient dataset containing all required features and labels.

```
[ ]: import pandas as pd

# Load preprocessed dataset from previous assignment
df = pd.read_csv(my_file_path + "preprocessing_generate_patient_dataset.csv")

# Display basic info and preview
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6501 entries, 0 to 6500
Data columns (total 12 columns):
#   Column                                Non-Null Count  Dtype
---  -
0   patient_id                            6501 non-null   object
1   timestamp                             6501 non-null   object
2   clinical_note                         6501 non-null   object
3   temperature_zscore                   6501 non-null   float64
4   heart_rate_zscore                    6501 non-null   float64
5   blood_glucose_zscore                 6501 non-null   float64
6   oxygen_saturation_zscore             6501 non-null   float64
7   systolic_bp_zscore                   6501 non-null   float64
8   diastolic_bp_zscore                  6501 non-null   float64
9   weight_zscore                        6501 non-null   float64
10  note_status                           6501 non-null   object
11  note_status_encoded                   6501 non-null   int64
dtypes: float64(7), int64(1), object(4)
memory usage: 609.6+ KB
None
```

```
[ ]: print(df.head())
```

```
   patient_id  timestamp  clinical_note \
0      P0001  2025-01-01  Patient stable post-surgery. Vitals within nor...
1      P0001  2025-01-02  Mildly elevated heart rate and temperature. Mo...
2      P0001  2025-01-03  Temperature trending upward. Possible low-grad...
3      P0001  2025-01-04  Temperature stabilizing. Patient reports impro...
4      P0001  2025-01-05  Patient showing signs of steady recovery. Vita...
```

	temperature_zscore	heart_rate_zscore	blood_glucose_zscore	\
0	0.360442	0.256585	-0.617148	
1	1.089682	0.881403	-0.438365	
2	2.548162	1.506221	-0.736337	
3	1.818922	0.673130	-0.140392	
4	0.725062	0.048312	-0.855527	

	oxygen_saturation_zscore	systolic_bp_zscore	diastolic_bp_zscore	\
0	0.029323	1.034085	0.690185	
1	-0.785334	1.423315	1.208257	
2	-1.599992	1.812545	1.726329	
3	-0.785334	1.423315	1.467293	
4	0.029323	0.644855	0.690185	

	weight_zscore	note_status	note_status_encoded
0	-0.633036	Stable	3
1	-0.734106	Recovering	2
2	-0.784640	Recovering	2
3	-0.835175	Stable	3
4	-0.835175	Recovering	2

```
[ ]: print(df.describe())
```

	temperature_zscore	heart_rate_zscore	blood_glucose_zscore	\
count	6.501000e+03	6.501000e+03	6.501000e+03	
mean	-5.744674e-15	-5.027683e-17	3.541237e-16	
std	1.000077e+00	1.000077e+00	1.000077e+00	
min	-3.285759e+00	-3.075779e+00	-2.107011e+00	
25%	-7.334186e-01	-5.765064e-01	-8.555265e-01	
50%	-4.178464e-03	-1.599609e-01	-2.595813e-01	
75%	3.604416e-01	6.731301e-01	9.323091e-01	
max	6.923603e+00	6.088221e+00	4.627169e+00	

	oxygen_saturation_zscore	systolic_bp_zscore	diastolic_bp_zscore	\
count	6.501000e+03	6.501000e+03	6.501000e+03	
mean	3.319364e-15	2.972891e-16	-1.141065e-15	
std	1.000077e+00	1.000077e+00	1.000077e+00	
min	-6.487937e+00	-3.247445e+00	-3.713428e+00	
25%	-7.853344e-01	-9.120650e-01	-8.640312e-01	
50%	2.932316e-02	-1.336051e-01	-8.692311e-02	
75%	8.439807e-01	6.448548e-01	6.901850e-01	
max	2.473296e+00	6.288689e+00	5.352834e+00	

	weight_zscore	note_status_encoded
count	6.501000e+03	6501.000000
mean	-7.290140e-16	2.432241
std	1.000077e+00	0.677030
min	-2.704969e+00	0.000000

25%	-6.835706e-01	2.000000
50%	4.918627e-02	3.000000
75%	2.765936e-01	3.000000
max	5.431159e+00	3.000000

## Step 2: Data Structure and Integrity Check

Check data types, confirm absence of missing values, and review main variables.

```
[ ]: # Check data info and missing values
print(df.info())
print(df.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6501 entries, 0 to 6500
Data columns (total 12 columns):
#   Column                Non-Null Count  Dtype
---  -
0   patient_id            6501 non-null  object
1   timestamp              6501 non-null  object
2   clinical_note          6501 non-null  object
3   temperature_zscore     6501 non-null  float64
4   heart_rate_zscore      6501 non-null  float64
5   blood_glucose_zscore   6501 non-null  float64
6   oxygen_saturation_zscore 6501 non-null  float64
7   systolic_bp_zscore     6501 non-null  float64
8   diastolic_bp_zscore    6501 non-null  float64
9   weight_zscore          6501 non-null  float64
10  note_status            6501 non-null  object
11  note_status_encoded     6501 non-null  int64
dtypes: float64(7), int64(1), object(4)
memory usage: 609.6+ KB
None
patient_id            0
timestamp              0
clinical_note          0
temperature_zscore     0
heart_rate_zscore      0
blood_glucose_zscore   0
oxygen_saturation_zscore 0
systolic_bp_zscore     0
diastolic_bp_zscore    0
weight_zscore          0
note_status            0
note_status_encoded    0
dtype: int64
```

## Step 3: Check Label Distribution

Review the distribution of the clinical status label (note\_status) to ensure it is suitable for modeling.

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

plt.figure(figsize=(8, 5))
ax = sns.countplot(
    data=df,
    x="note_status",
    order=df["note_status"].value_counts().index,
    palette="pastel"
)

plt.title("Distribution of Clinical Note Status Labels", fontsize=14)
plt.xlabel("Note Status", fontsize=12)
plt.ylabel("Number of Records", fontsize=12)
plt.grid(axis="y", linestyle="--", alpha=0.5)
plt.tight_layout()

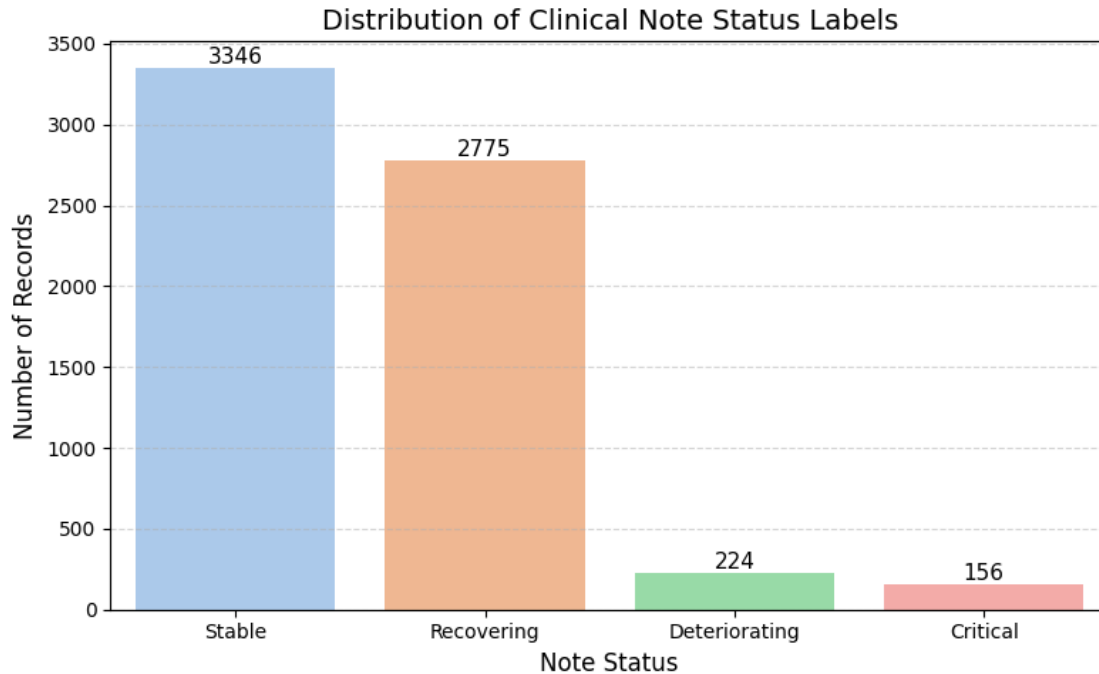
# Add count labels on each bar
for p in ax.patches:
    count = int(p.get_height())
    ax.annotate(f"{count}", (p.get_x() + p.get_width() / 2, p.get_height()),
                ha="center", va="bottom", fontsize=11, color="black")

plt.show()
```

<ipython-input-11-2168602507>:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

```
ax = sns.countplot(
```



#### Step 4: Standardize Note Status Label Encoding

Map the clinical status label (`note_status`) to standardized integer codes: 0 for Stable, 1 for Recovering, 2 for Deteriorating, and 3 for Critical. Replace the existing `note_status_encoded` with these values for consistency in downstream modeling.

```
[ ]: # Define standardized label mapping
note_status_mapping = {
    "Stable": 0,
    "Recovering": 1,
    "Deteriorating": 2,
    "Critical": 3
}

# Apply mapping to the note_status column and overwrite note_status_encoded
df['note_status_encoded'] = df['note_status'].map(note_status_mapping)

# Preview mapping results
print(df[['note_status', 'note_status_encoded']].head())
print(df['note_status_encoded'].value_counts().sort_index())
```

	note_status	note_status_encoded
0	Stable	0
1	Recovering	1
2	Recovering	1
3	Stable	0



```

4 Recovering                                1
note_status_encoded
0      3346
1      2775
2       224
3       156
Name: count, dtype: int64

```

## Summary

The preprocessed patient dataset has been successfully loaded and validated. All core variables, including vital sign z-scores, clinical notes, and coded clinical status labels, are present with no missing values. The distribution of the “note\_status” label has been visualized, revealing the class imbalance issue that will be addressed in the modeling phase. After confirming the dataset structure and quality, we are ready for AI-driven NLP feature engineering and predictive modeling.

## 3 2. NLP Feature Engineering

### Step 1: Sentiment Analysis on Clinical Notes

Extract sentiment label and score from each clinical note using a transformer-based sentiment analysis model.

```

[ ]: from transformers import pipeline
      from tqdm import tqdm
      import pandas as pd

      sentiment_pipe = pipeline("sentiment-analysis",
                                ↪model="distilbert-base-uncased-finetuned-sst-2-english")

      def get_sentiment(text):
          result = sentiment_pipe(text[:512])[0]
          return pd.Series([result['label'], result['score']])

      tqdm.pandas(desc="Sentiment Analysis")
      df[['sentiment_label', 'sentiment_score']] = df['clinical_note'].
          ↪progress_apply(get_sentiment)

```

/usr/local/lib/python3.11/dist-packages/huggingface\_hub/utils/\_auth.py:94:

UserWarning:

The secret `HF\_TOKEN` does not exist in your Colab secrets.

To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/tokens>), set it as secret in your Google Colab and restart your session.

You will be able to reuse this secret in all of your notebooks.

Please note that authentication is recommended but still optional to access public models or datasets.

warnings.warn(

Device set to use cuda:0

Sentiment Analysis: 0%| | 2/6501 [00:00<13:08, 8.24it/s] You seem to be using the pipelines sequentially on GPU. In order to maximize efficiency please use a dataset  
Sentiment Analysis: 100%| | 6501/6501 [00:32<00:00, 198.96it/s]

```
[ ]: # Preview sentiment features
print(df[['sentiment_label', 'sentiment_score']].head())
```

	sentiment_label	sentiment_score
0	POSITIVE	0.996446
1	NEGATIVE	0.970783
2	POSITIVE	0.873627
3	NEGATIVE	0.983538
4	POSITIVE	0.990160

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

plt.figure(figsize=(8, 5))
ax = sns.countplot(
    data=df,
    x="sentiment_label",
    order=df["sentiment_label"].value_counts().index,
    palette="pastel"
)

plt.title("Distribution of Sentiment Labels", fontsize=14)
plt.xlabel("Sentiment Label", fontsize=12)
plt.ylabel("Number of Records", fontsize=12)
plt.grid(axis="y", linestyle="--", alpha=0.5)
plt.tight_layout()

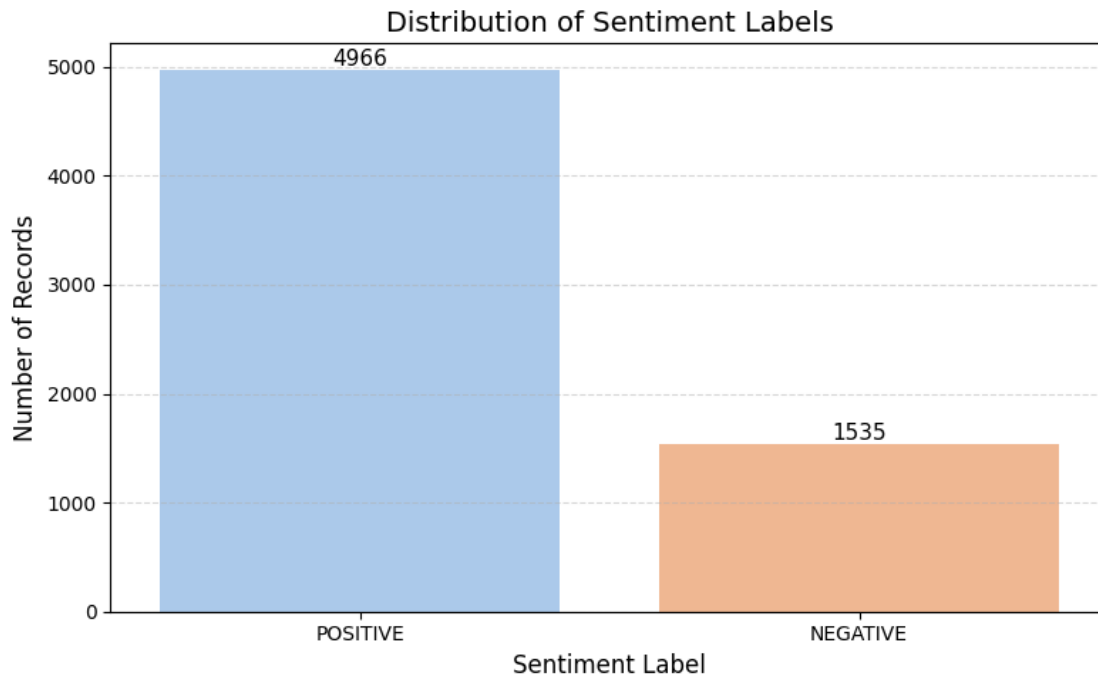
# Add count labels on each bar
for p in ax.patches:
    count = int(p.get_height())
    ax.annotate(f"{count}", (p.get_x() + p.get_width() / 2, p.get_height()),
                ha="center", va="bottom", fontsize=11, color="black")

plt.show()
```

<ipython-input-15-1878399755>:5: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

```
ax = sns.countplot(
```



To check for ambiguous clinical notes, I analyze the distribution of sentiment scores. Scores near 0.5 suggest uncertainty, while scores closer to 0 or 1 indicate confident classification.

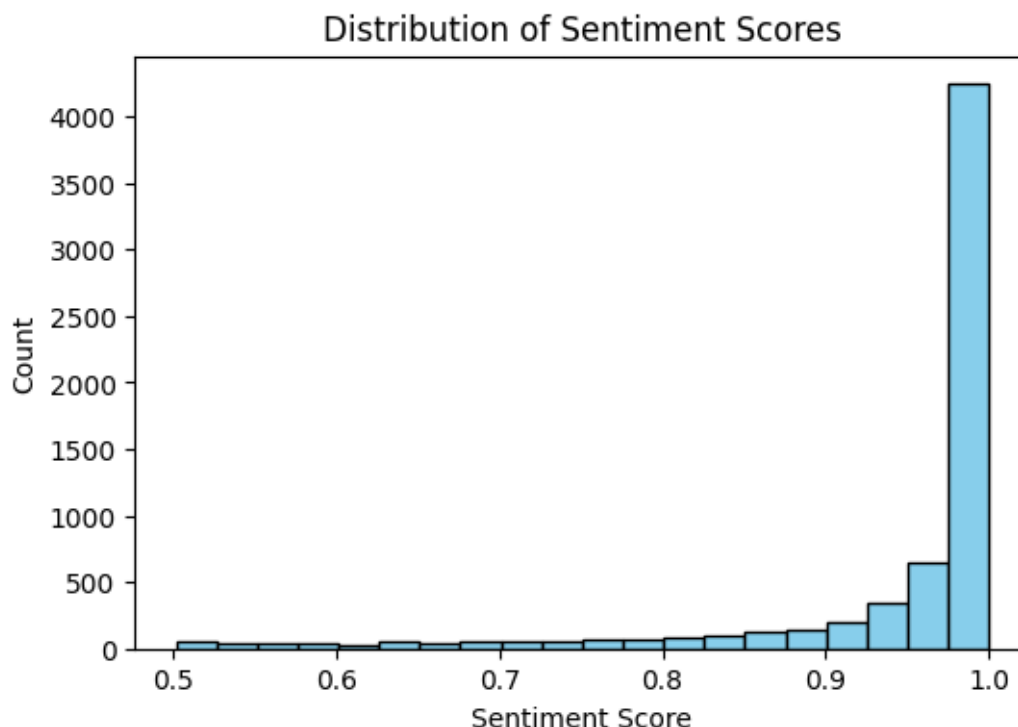
```
[ ]: # Count the number of potential "neutral" cases (sentiment_score between 0.4
    ↪and 0.6)
neutral_count = ((df['sentiment_score'] >= 0.4) & (df['sentiment_score'] <= 0.
    ↪6)).sum()
total = len(df)
percent_neutral = 100 * neutral_count / total

print(f"Potential 'neutral' cases (score between 0.4 and 0.6): {neutral_count}
    ↪({percent_neutral:.2f}%")
```

Potential 'neutral' cases (score between 0.4 and 0.6): 184 (2.83%)

```
[ ]: # Visualize the distribution of sentiment scores for all clinical notes
import matplotlib.pyplot as plt

plt.figure(figsize=(6,4))
plt.hist(df['sentiment_score'], bins=20, color='skyblue', edgecolor='black')
plt.title("Distribution of Sentiment Scores")
plt.xlabel("Sentiment Score")
plt.ylabel("Count")
plt.show()
```



```
[ ]: # Preview random samples of "neutral" sentiment cases
neutral_samples = df[(df['sentiment_score'] >= 0.4) & (df['sentiment_score'] <= 0.6)].sample(5)
print(neutral_samples[['clinical_note', 'note_status', 'sentiment_label', 'sentiment_score']])
```

	clinical_note	note_status \
1524	Weight omitted today. Patient continues to exh...	Stable
2698	Final day of monitoring. Patient ready for dis...	Stable
5205	Mild tachycardia and elevated temperature note...	Deteriorating
3905	Patient remains stable; slight improvement in ...	Stable
6489	Patient stable post-surgery. Vitals within nor...	Stable

	sentiment_label	sentiment_score
1524	NEGATIVE	0.548620
2698	NEGATIVE	0.583013
5205	NEGATIVE	0.511570
3905	POSITIVE	0.512316
6489	POSITIVE	0.577934

The sentiment scores are strongly polarized, with very few notes falling in the ambiguous range (0.4–0.6). Only 2.8% of clinical notes show unclear sentiment, confirming that binary sentiment labels are appropriate for this dataset.

## Step 2: Encode Sentiment Labels

Convert the sentiment labels into binary numeric values for downstream modeling. Encode sentiment labels as binary values (1=POSITIVE, 0=NEGATIVE) for model input.

```
[ ]: # Encode sentiment label as binary (POSITIVE=1, NEGATIVE=0)
df['sentiment_label_encoded'] = df['sentiment_label'].map({'POSITIVE': 1,
↳ 'NEGATIVE': 0})

# Preview encoded results
print(df[['sentiment_label', 'sentiment_label_encoded']].head())
print(df['sentiment_label_encoded'].value_counts())
```

```
sentiment_label  sentiment_label_encoded
0      POSITIVE                      1
1     NEGATIVE                      0
2      POSITIVE                      1
3     NEGATIVE                      0
4      POSITIVE                      1
sentiment_label_encoded
1      4966
0      1535
Name: count, dtype: int64
```

## Step 3: Generate MiniLM Embeddings for Clinical Notes

Convert each clinical note into a dense vector using the MiniLM model, creating numerical features that capture the semantic meaning of the text.

```
[ ]: from transformers import AutoTokenizer, AutoModel
import torch
import numpy as np
from tqdm import tqdm

# Load MiniLM model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("sentence-transformers/
↳ all-MiniLM-L6-v2")
model = AutoModel.from_pretrained("sentence-transformers/all-MiniLM-L6-v2")

# Function to get mean-pooled sentence embedding
def get_embedding(text):
    inputs = tokenizer(text[:512], return_tensors="pt", truncation=True,
↳ padding=True, max_length=64)
    with torch.no_grad():
        emb = model(**inputs).last_hidden_state.mean(dim=1).squeeze().cpu().
↳ numpy()
    return emb

# Generate embeddings with progress bar
```

```

embeddings = []
for note in tqdm(df['clinical_note'], desc="Generating MiniLM Embeddings"):
    embeddings.append(get_embedding(note))
embeddings = np.vstack(embeddings)

```

Generating MiniLM Embeddings: 100% | 6501/6501 [01:01<00:00, 105.49it/s]

```

[ ]: # Convert to DataFrame and merge with main df
embeddings_df = pd.DataFrame(embeddings, columns=[f'embedding_{i+1}' for i in
    ↪range(embeddings.shape[1])])
df = pd.concat([df.reset_index(drop=True), embeddings_df], axis=1)

# Preview embedding features
print(embeddings_df.shape)
print(embeddings_df.head())

```

(6501, 384)

	embedding_1	embedding_2	embedding_3	embedding_4	embedding_5	\
0	-0.002227	0.054226	-0.198225	0.007124	-0.314499	
1	0.049453	0.058550	0.163853	0.278248	-0.112794	
2	-0.074871	-0.096136	0.306630	0.355880	0.115569	
3	-0.002369	-0.074267	0.152957	0.536647	-0.151257	
4	0.005790	0.035201	0.092588	0.339710	-0.237124	

	embedding_6	embedding_7	embedding_8	embedding_9	embedding_10	...	\
0	-0.064337	-0.051729	0.139015	-0.080340	-0.160634	...	
1	-0.140065	-0.007666	0.156703	-0.058688	-0.029590	...	
2	-0.017233	0.000656	0.174866	-0.035941	0.209080	...	
3	-0.057069	0.056320	-0.003020	-0.189366	0.162416	...	
4	0.011571	-0.100337	0.100693	-0.095893	-0.358720	...	

	embedding_375	embedding_376	embedding_377	embedding_378	embedding_379	\
0	-0.185589	0.169756	0.236088	0.006934	0.170186	
1	-0.494870	0.072143	-0.035751	0.089070	0.000560	
2	-0.375889	-0.122558	-0.029090	0.086142	-0.097378	
3	-0.586711	-0.131158	0.086496	0.063815	0.236612	
4	-0.379787	0.157074	-0.139231	0.180917	-0.043465	

	embedding_380	embedding_381	embedding_382	embedding_383	embedding_384
0	0.059771	0.004493	0.054284	0.086865	-0.191795
1	0.226057	0.091370	-0.274222	-0.128617	0.189073
2	0.088276	-0.063335	-0.246799	-0.399010	0.108029
3	0.215968	-0.039085	-0.395321	-0.270403	0.138713
4	-0.022627	0.010339	0.045971	0.056790	-0.261534

[5 rows x 384 columns]

## 4 3. Feature and Target Assignment

### Step 1: Select Features for Modeling

Combine structured vital signs, sentiment analysis features, and MiniLM embeddings to form the initial input feature set.

```
[ ]: # List of structured vital sign features
vital_features = [
    'temperature_zscore', 'heart_rate_zscore', 'blood_glucose_zscore',
    'oxygen_saturation_zscore', 'systolic_bp_zscore', 'diastolic_bp_zscore',
    'weight_zscore'
]

# NLP features (sentiment + embeddings)
nlp_features = ['sentiment_label_encoded', 'sentiment_score'] +
    [f'embedding_{i+1}' for i in range(embeddings_df.shape[1])]

# Combine all features for model input
feature_cols = vital_features + nlp_features
X = df[feature_cols]
print("Feature matrix shape:", X.shape)
```

Feature matrix shape: (6501, 393)

### Step 2: Define Target Variable

Set the encoded clinical status label as the prediction target for multi-class classification.

```
[ ]: # Target variable (multi-class clinical status)
y = df['note_status_encoded']
print("Target distribution:\n", y.value_counts().sort_index())
```

```
Target distribution:
note_status_encoded
0      3346
1      2775
2       224
3       156
Name: count, dtype: int64
```

## 5 4. Train-Test Split

### Step 1: Split the Dataset

Split the dataset into training and test sets, stratifying by the target variable to maintain class distribution.

```
[ ]: from sklearn.model_selection import train_test_split
```

```

# 80% for training, 20% for testing, stratified by class
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

print("Training set shape:", X_train.shape)
print("Test set shape:", X_test.shape)
print("Training target distribution:\n", y_train.value_counts().sort_index())
print("Test target distribution:\n", y_test.value_counts().sort_index())

```

Training set shape: (5200, 393)

Test set shape: (1301, 393)

Training target distribution:

note\_status\_encoded

0 2676

1 2220

2 179

3 125

Name: count, dtype: int64

Test target distribution:

note\_status\_encoded

0 670

1 555

2 45

3 31

Name: count, dtype: int64

## 6 5. Class Imbalance Handling (SMOTE)

### Step 1: Balance the Training Set with SMOTE

Apply SMOTE to the training set to generate synthetic minority samples and balance class distribution. The process may take some time for high-dimensional data.

```

[ ]: from imblearn.over_sampling import SMOTE
from collections import Counter
import time
from tqdm import tqdm

# Optional: label mapping for pretty output
note_status_mapping = {0: "Stable", 1: "Recovering", 2: "Deteriorating", 3: "Critical"}

# Print original class distribution (with labels)
print("Original training class distribution:")
for k, v in Counter(y_train).items():
    print(f" {note_status_mapping[k]} ({k}): {v}")

```



```

# Start timer
start = time.time()

# Run SMOTE with overall progress feel
print("Applying SMOTE to balance classes...")
for _ in tqdm(range(1), desc="SMOTE Oversampling"):
    sm = SMOTE(random_state=42)
    X_train_bal, y_train_bal = sm.fit_resample(X_train, y_train)

# End timer
end = time.time()
print(f"SMOTE completed in {end-start:.2f} seconds.")

```

Original training class distribution:

```

Stable (0): 2676
Recovering (1): 2220
Critical (3): 125
Deteriorating (2): 179

```

Applying SMOTE to balance classes...

```

SMOTE Oversampling: 100%|          | 1/1 [00:00<00:00,  6.31it/s]

```

SMOTE completed in 0.16 seconds.

```

[ ]: # Print balanced class distribution (with labels)
print("Balanced training class distribution:")
for k, v in Counter(y_train_bal).items():
    print(f" {note_status_mapping[k]} ({k}): {v}")

print("Balanced training set shape:", X_train_bal.shape)

```

Balanced training class distribution:

```

Stable (0): 2676
Recovering (1): 2676
Critical (3): 2676
Deteriorating (2): 2676

```

Balanced training set shape: (10704, 393)

## 7 6. Model Development and Evaluation

### Step 1: Step 1: Define the Traditional Model Evaluation Function

Define a general-purpose evaluation function for traditional models, reporting metrics, confusion matrix, and classification report.

```
[ ]: def evaluate_model(model, X_train, y_train, X_test, y_test, model_name="Model"):
    """
    Evaluate a classification model with standard metrics and visualize the
    ↪confusion matrix.
    Returns a summary dictionary for results table, including raw evaluation
    ↪artifacts for LLM analysis.
    """
    from sklearn.metrics import (
        accuracy_score, f1_score, precision_score, recall_score,
        classification_report, confusion_matrix
    )
    import matplotlib.pyplot as plt
    import seaborn as sns

    # Predict
    y_pred = model.predict(X_test)
    y_train_pred = model.predict(X_train)

    # Metrics
    acc = accuracy_score(y_test, y_pred)
    f1_macro = f1_score(y_test, y_pred, average="macro")
    prec_macro = precision_score(y_test, y_pred, average="macro")
    recall_macro = recall_score(y_test, y_pred, average="macro")
    f1_train = f1_score(y_train, y_train_pred, average="macro")

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)

    # Print results
    print(f"\n===== {model_name} Evaluation =====")
    print(f"Accuracy: {acc:.4f} | Macro F1: {f1_macro:.4f} | Precision:
    ↪{prec_macro:.4f} | Recall: {recall_macro:.4f}")

    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, cmap="Blues", fmt="d")
    plt.title(f"{model_name} Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.tight_layout()
    plt.show()

    print("\nClassification Report:")
    print(classification_report(y_test, y_pred, digits=3))

    # Add these for LLM-friendly result summaries
    cm_list = cm.tolist() # For JSON/prompt/LLM
```

```

    report_dict = classification_report(y_test, y_pred, digits=3,
    ↪output_dict=True) # For AI parsing
    report_str = classification_report(y_test, y_pred, digits=3) # For human
    ↪reading

    # Return summary dict
    return {
        "Model": model_name,
        "Train F1": round(f1_train, 4),
        "Test F1": round(f1_macro, 4),
        "Accuracy": round(acc, 4),
        "Precision": round(prec_macro, 4),
        "Recall": round(recall_macro, 4),
        "Confusion Matrix": cm_list,
        "Classification Report (dict)": report_dict,
        "Classification Report (str)": report_str
    }

```

## Step 2: Train and Evaluate Random Forest

Train a Random Forest classifier. Evaluate its test set performance using key metrics and visualize the confusion matrix.

```

[ ]: import warnings
warnings.filterwarnings('ignore')
from sklearn.ensemble import RandomForestClassifier

results_list = []

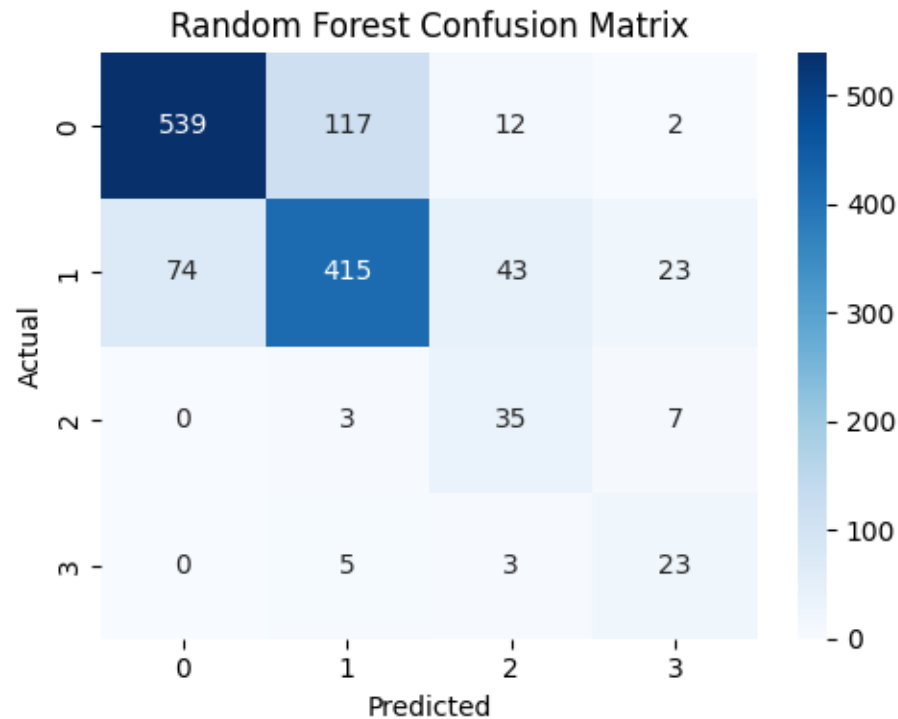
# Train Random Forest on balanced training set
rf = RandomForestClassifier(
    n_estimators=300,
    max_depth=10,
    min_samples_split=8,
    min_samples_leaf=4,
    max_features='sqrt',
    class_weight='balanced',
    random_state=42,
    n_jobs=-1
)
rf.fit(X_train_bal, y_train_bal)

# Evaluate model performance on test set
rf_results = evaluate_model(rf, X_train_bal, y_train_bal, X_test, y_test,
    ↪model_name="Random Forest")
results_list.append(rf_results)

```

==== Random Forest Evaluation ====

Accuracy: 0.7779 | Macro F1: 0.6601 | Precision: 0.6106 | Recall: 0.7680



Classification Report:

	precision	recall	f1-score	support
0	0.879	0.804	0.840	670
1	0.769	0.748	0.758	555
2	0.376	0.778	0.507	45
3	0.418	0.742	0.535	31
accuracy			0.778	1301
macro avg	0.611	0.768	0.660	1301
weighted avg	0.804	0.778	0.786	1301

### Step 3: Train and Evaluate XGBoost

Train an XGBoost classifier using the same features and balanced data. Evaluate its performance using the same metrics for fair comparison.

```
[ ]: from xgboost import XGBClassifier  
  
# Train XGBoost on balanced training set
```

```

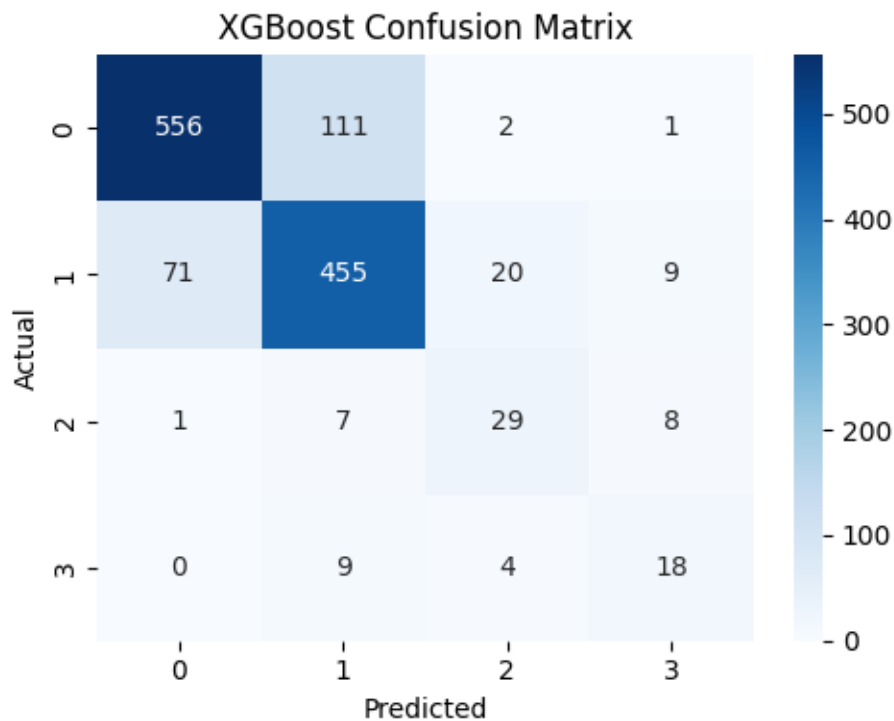
xgb = XGBClassifier(
    n_estimators=300,
    max_depth=6,
    learning_rate=0.07,
    subsample=0.8,
    colsample_bytree=0.8,
    min_child_weight=6,
    gamma=2,
    use_label_encoder=False,
    eval_metric='mlogloss',
    random_state=42,
    n_jobs=-1
)
xgb.fit(X_train_bal, y_train_bal)

# Evaluate model performance on test set
xgb_results = evaluate_model(xgb, X_train_bal, y_train_bal, X_test, y_test,
    ↪model_name="XGBoost")
results_list.append(xgb_results)

```

===== XGBoost Evaluation =====

Accuracy: 0.8132 | Macro F1: 0.6936 | Precision: 0.6736 | Recall: 0.7187



#### Classification Report:

	precision	recall	f1-score	support
0	0.885	0.830	0.857	670
1	0.782	0.820	0.800	555
2	0.527	0.644	0.580	45
3	0.500	0.581	0.537	31
accuracy			0.813	1301
macro avg	0.674	0.719	0.694	1301
weighted avg	0.820	0.813	0.815	1301

#### Step 4: Train and Evaluate MLP Neural Network

Train a Multi-Layer Perceptron (MLP) neural network. Evaluate and visualize its classification performance.

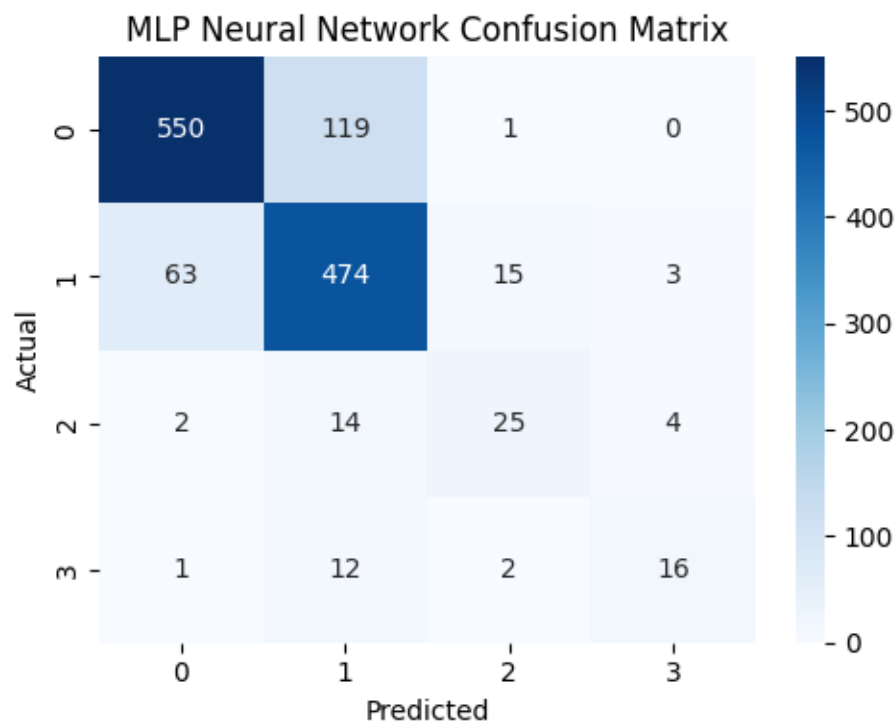
```
[ ]: from sklearn.neural_network import MLPClassifier

# Train Multi-layer Perceptron on balanced training set
mlp = MLPClassifier(
    hidden_layer_sizes=(128, 64),
    activation='relu',
    solver='adam',
    alpha=0.01,
    batch_size=64,
    learning_rate_init=0.002,
    max_iter=200,
    early_stopping=True,
    random_state=42
)
mlp.fit(X_train_bal, y_train_bal)

# Evaluate model performance on test set
mlp_results = evaluate_model(mlp, X_train_bal, y_train_bal, X_test, y_test,
    ↪model_name="MLP Neural Network")
results_list.append(mlp_results)
```

===== MLP Neural Network Evaluation =====

Accuracy: 0.8186 | Macro F1: 0.7059 | Precision: 0.7339 | Recall: 0.6867



Classification Report:

	precision	recall	f1-score	support
0	0.893	0.821	0.855	670
1	0.766	0.854	0.807	555
2	0.581	0.556	0.568	45
3	0.696	0.516	0.593	31
accuracy			0.819	1301
macro avg	0.734	0.687	0.706	1301
weighted avg	0.823	0.819	0.819	1301

### Step 5: Define the Transformer Model Evaluation Function

Define a specialized evaluation function for transformer-based models, capturing all metrics and artifacts needed for LLM-assisted interpretation.

```
[ ]: from sklearn.model_selection import train_test_split

# Split data into train and test sets
train_texts, test_texts, train_labels_raw, test_labels_raw = train_test_split(
    df["clinical_note"].tolist(),
    df["note_status"].tolist(),
```

```

    test_size=0.2,
    random_state=42
)

# Manual label mapping for strict order: 0=Stable, 1=Recovering,
↪2=Deteriorating, 3=Critical
status2id = {
    "Stable": 0,
    "Recovering": 1,
    "Deteriorating": 2,
    "Critical": 3
}
id2status = {v: k for k, v in status2id.items()}

# Map original labels to integer labels
train_labels = [status2id[x] for x in train_labels_raw]
test_labels = [status2id[x] for x in test_labels_raw]

print("Label mapping:", status2id)
print("Train label unique values:", set(train_labels))
print("Test label unique values:", set(test_labels))

```

Label mapping: {'Stable': 0, 'Recovering': 1, 'Deteriorating': 2, 'Critical': 3}  
Train label unique values: {0, 1, 2, 3}  
Test label unique values: {0, 1, 2, 3}

```

[ ]: import numpy as np
from sklearn.metrics import (
    accuracy_score, f1_score, precision_score, recall_score, confusion_matrix,
↪classification_report
)
import matplotlib.pyplot as plt
import seaborn as sns

def evaluate_transformer_model(y_true, y_pred, model_name="Model",
↪target_names=None):
    """
    Evaluate a transformer model's predictions and print confusion matrix and
↪metrics.

    Returns summary dictionary for results table, including confusion matrix
↪and classification report.
    """
    acc = accuracy_score(y_true, y_pred)
    f1_macro = f1_score(y_true, y_pred, average="macro")
    prec_macro = precision_score(y_true, y_pred, average="macro")
    recall_macro = recall_score(y_true, y_pred, average="macro")

```



```

cm = confusion_matrix(y_true, y_pred)

print(f"\n===== {model_name} Evaluation =====")
print(f"Accuracy: {acc:.4f} | Macro F1: {f1_macro:.4f} | Precision:␣
↪{prec_macro:.4f} | Recall: {recall_macro:.4f}")

plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, cmap="Blues", fmt="d")
plt.title(f"{model_name} Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()

# Classification reports
report_dict = classification_report(y_true, y_pred,␣
↪target_names=target_names, digits=3, output_dict=True)
report_str = classification_report(y_true, y_pred,␣
↪target_names=target_names, digits=3)
print("\nClassification Report:")
print(report_str)

return {
    "Model": model_name,
    "Test F1": round(f1_macro, 4),
    "Accuracy": round(acc, 4),
    "Precision": round(prec_macro, 4),
    "Recall": round(recall_macro, 4),
    "Confusion Matrix": cm.tolist(),                # <-- For saving or LLM␣
↪prompt
    "Classification Report (dict)": report_dict,    # <-- For LLM, code,␣
↪summary
    "Classification Report (str)": report_str        # <-- For direct prompt/
↪human reading
}

```

```

[ ]: from transformers import AutoTokenizer, AutoModelForSequenceClassification,␣
↪Trainer, TrainingArguments
from datasets import Dataset
import os

os.environ["WANDB_DISABLED"] = "true" # Disable external logging

def train_and_predict_transformer(model_ckpt, train_texts, train_labels,␣
↪test_texts, test_labels, model_name="Transformer"):
    # 1. Load tokenizer and model

```

```

tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModelForSequenceClassification.from_pretrained(model_ckpt,
↳ num_labels=4)

# 2. Tokenize texts
train_encodings = tokenizer(train_texts, truncation=True, padding=True)
test_encodings = tokenizer(test_texts, truncation=True, padding=True)

train_dataset = Dataset.from_dict(**train_encodings, "label":
↳ train_labels})
test_dataset = Dataset.from_dict(**test_encodings, "label": test_labels})

# 3. Training arguments
training_args = TrainingArguments(
    output_dir="./results",
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=5,
    learning_rate=2e-5,
    weight_decay=0.01,
    warmup_ratio=0.1,
    logging_dir="./logs",
    logging_strategy="epoch",
    save_strategy="no",
    report_to="none"
)

# 4. Train
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset
)
trainer.train()

# 5. Predict
preds = trainer.predict(test_dataset)
y_pred = np.argmax(preds.predictions, axis=1)
return y_pred

```

## Step 6: Train and Evaluate BERT-base Transformer

Fine-tune a BERT-base transformer on clinical note text for health status prediction. Evaluate its test performance with standard metrics and confusion matrix.

```
[ ]: target_names = ['Stable', 'Recovering', 'Deteriorating', 'Critical']

# 1. BERT-base-uncased
bert_pred = train_and_predict_transformer("bert-base-uncased", train_texts,
    ↪ train_labels, test_texts, test_labels, model_name="BERT-base")
results_list.append(evaluate_transformer_model(test_labels, bert_pred,
    ↪ "BERT-base", target_names=target_names))
```

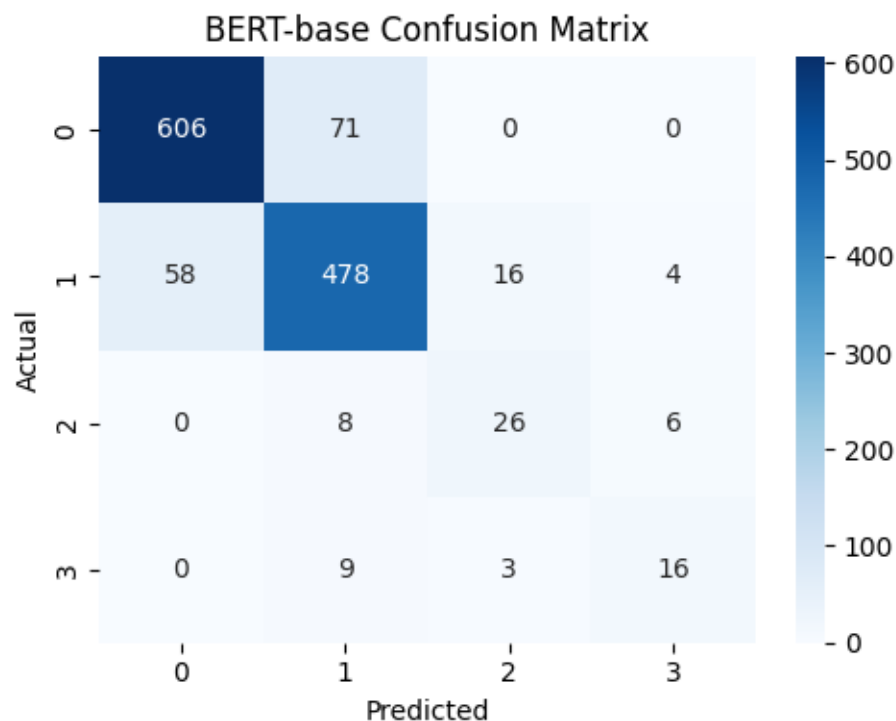
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized:  
['classifier.bias', 'classifier.weight']  
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

===== BERT-base Evaluation =====

Accuracy: 0.8655 | Macro F1: 0.7401 | Precision: 0.7376 | Recall: 0.7441



Classification Report:

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

Stable	0.913	0.895	0.904	677
Recovering	0.845	0.860	0.852	556
Deteriorating	0.578	0.650	0.612	40
Critical	0.615	0.571	0.593	28
accuracy			0.865	1301
macro avg	0.738	0.744	0.740	1301
weighted avg	0.867	0.865	0.866	1301

## Step 7: Train and Evaluate BioBERT Transformer

Fine-tune a BioBERT transformer model on the same prediction task. Assess its results using identical metrics for comparison.

```
[ ]: # 2. BioBERT
biobert_pred = train_and_predict_transformer("dmis-lab/biobert-base-cased-v1.
↪1", train_texts, train_labels, test_texts, test_labels, model_name="BioBERT")
results_list.append(evaluate_transformer_model(test_labels, biobert_pred, ↪
↪"BioBERT", target_names=target_names))
```

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at dmis-lab/biobert-base-cased-v1.1 and are newly initialized:

```
['classifier.bias', 'classifier.weight']
```

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

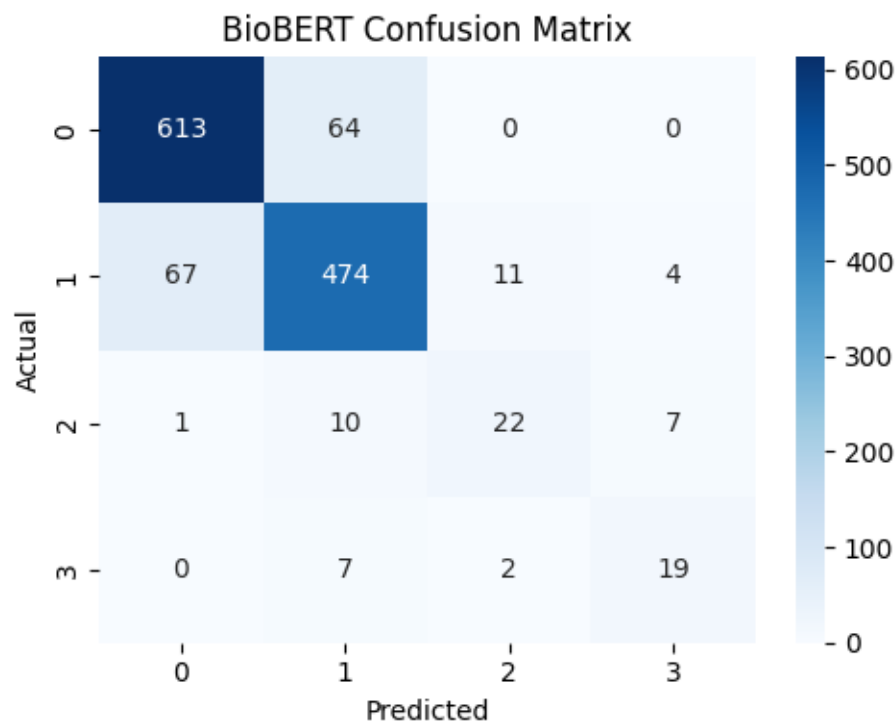
Asking to truncate to max\_length but no maximum length is provided and the model has no predefined maximum length. Default to no truncation.

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

===== BioBERT Evaluation =====

Accuracy: 0.8670 | Macro F1: 0.7495 | Precision: 0.7540 | Recall: 0.7466



#### Classification Report:

	precision	recall	f1-score	support
Stable	0.900	0.905	0.903	677
Recovering	0.854	0.853	0.853	556
Deteriorating	0.629	0.550	0.587	40
Critical	0.633	0.679	0.655	28
accuracy			0.867	1301
macro avg	0.754	0.747	0.749	1301
weighted avg	0.866	0.867	0.867	1301

### Step 8: Train and Evaluate DeBERTa Transformer

Fine-tune a DeBERTa transformer model for the multi-class classification task. Evaluate and compare its predictive performance.

```
[ ]: # 3. DeBERTa
deberta_pred = train_and_predict_transformer("microsoft/deberta-base",
↪train_texts, train_labels, test_texts, test_labels, model_name="DeBERTa")
results_list.append(evaluate_transformer_model(test_labels, deberta_pred,
↪"DeBERTa", target_names=target_names))
```

Some weights of DebertaForSequenceClassification were not initialized from the model checkpoint at microsoft/deberta-base and are newly initialized:

['classifier.bias', 'classifier.weight', 'pooler.dense.bias',  
'pooler.dense.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

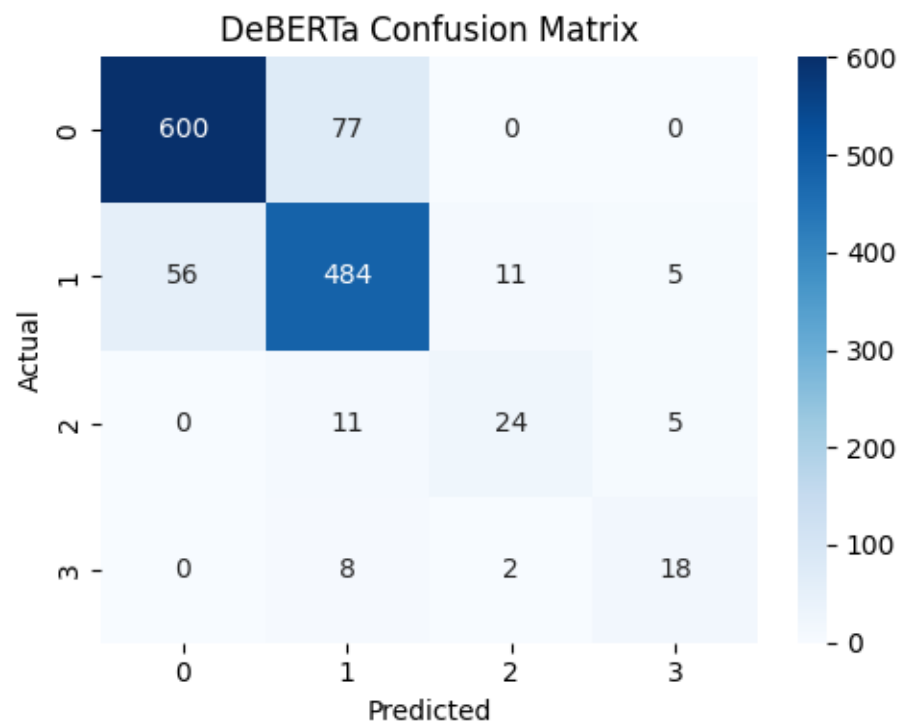
Asking to truncate to max\_length but no maximum length is provided and the model has no predefined maximum length. Default to no truncation.

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

===== DeBERTa Evaluation =====

Accuracy: 0.8655 | Macro F1: 0.7546 | Precision: 0.7602 | Recall: 0.7499



Classification Report:

	precision	recall	f1-score	support
Stable	0.915	0.886	0.900	677
Recovering	0.834	0.871	0.852	556
Deteriorating	0.649	0.600	0.623	40
Critical	0.643	0.643	0.643	28

accuracy			0.865	1301
macro avg	0.760	0.750	0.755	1301
weighted avg	0.866	0.865	0.866	1301

### Step 9: Summarize and Compare All Model Results

Aggregate all results into a summary table for direct comparison across all traditional and transformer-based models.

```
[ ]: import pandas as pd
```

```
summary_df = pd.DataFrame(results_list)
display(summary_df)
```

	Model	Train F1	Test F1	Accuracy	Precision	Recall	\
0	Random Forest	0.9686	0.6601	0.7779	0.6106	0.7680	
1	XGBoost	0.9902	0.6936	0.8132	0.6736	0.7187	
2	MLP Neural Network	0.9786	0.7059	0.8186	0.7339	0.6867	
3	BERT-base	NaN	0.7401	0.8655	0.7376	0.7441	
4	BioBERT	NaN	0.7495	0.8670	0.7540	0.7466	
5	DeBERTa	NaN	0.7546	0.8655	0.7602	0.7499	

	Confusion Matrix	\
0	[[539, 117, 12, 2], [74, 415, 43, 23], [0, 3, ...	
1	[[556, 111, 2, 1], [71, 455, 20, 9], [1, 7, 29...	
2	[[550, 119, 1, 0], [63, 474, 15, 3], [2, 14, 2...	
3	[[606, 71, 0, 0], [58, 478, 16, 4], [0, 8, 26,...	
4	[[613, 64, 0, 0], [67, 474, 11, 4], [1, 10, 22...	
5	[[600, 77, 0, 0], [56, 484, 11, 5], [0, 11, 24...	

	Classification Report (dict)	\
0	{'0': {'precision': 0.8792822185970636, 'recal...	
1	{'0': {'precision': 0.8853503184713376, 'recal...	
2	{'0': {'precision': 0.8928571428571429, 'recal...	
3	{'Stable': {'precision': 0.9126506024096386, '...	
4	{'Stable': {'precision': 0.9001468428781204, '...	
5	{'Stable': {'precision': 0.9146341463414634, '...	

	Classification Report (str)
0	precision recall f1-score ...
1	precision recall f1-score ...
2	precision recall f1-score ...
3	precision recall f1-score ...
4	precision recall f1-score ...
5	precision recall f1-score ...

## 8 7. LLM-Assisted Model Interpretation and Reporting

Leverage a Large Language Model (LLM) such as GPT-4o to automatically interpret, compare, and summarize the predictive performance of all evaluated models. This enables objective, human-readable scientific reporting and evidence-based model selection.

### Step 1: Prepare the Model Performance Summary Table

Convert the pandas summary table of all model results into a markdown-formatted string for easier consumption by an LLM.

```
[ ]: # Select key columns for summary and convert to markdown for LLM input
summary_table_text = summary_df[["Model", "Test F1", "Accuracy", "Precision",
    ↪ "Recall"]].to_markdown(index=False)
```

### Step 2: Define an Expert Prompt for the LLM

Write an instruction prompt that asks the LLM to analyze and summarize the model comparison table with scientific rigor and clarity.

```
[ ]: LLM_SUMMARY_PROMPT = """
You are an expert data scientist. Below is a summary table reporting key test_
    ↪ set performance metrics (Test F1, Accuracy, Precision, Recall) for six_
    ↪ machine learning models (three traditional and three transformer-based) on a_
    ↪ multi-class clinical status prediction task.

Summary Table (test set results):

{summary_table}

Instructions:
1. Compare the performance of traditional machine learning models (Random_
    ↪ Forest, XGBoost, MLP Neural Network) with transformer-based models_
    ↪ (BERT-base, BioBERT, DeBERTa), citing specific metrics and models by name.
2. Identify the best-performing model(s) and justify your conclusion with_
    ↪ numerical evidence.
3. Discuss interesting trends, weaknesses, or trade-offs, such as class_
    ↪ imbalance, computational resources, and overfitting risks.
4. Briefly comment on each model's practicality for clinical deployment,_
    ↪ considering real-world resource or interpretability constraints.
5. Suggest one area for further improvement or future research.
6. Conclude with a clear, formal academic recommendation (1-2 sentences).

Write your summary in concise, formal, and academic English, suitable for a_
    ↪ scientific report. Use bullet points if appropriate for clarity.
"""
```

### Step 3: Format and Compose the Final Prompt

Insert the model performance table into the prompt template for LLM processing.



```
[ ]: # Merge the prompt with the actual model performance table
final_prompt = LLM_SUMMARY_PROMPT.format(summary_table=summary_table_text)
```

#### Step 4: Generate an Expert Summary via LLM

Send the formatted prompt to your LLM API (e.g., Azure, OpenAI GPT-4o) and print the summary for reporting.

```
[ ]: # Call your LLM (replace with your actual LLM function, e.g., OpenAI/Azure call)
llm_response = model_prompt(final_prompt, system_prompt="You are an expert_
↳clinical data scientist. Write in academic style.")

# Output the summary for inclusion in your report
print(llm_response)
```

#### ### Comparative Analysis of Model Performance

##### 1. \*\*Performance Comparison: Traditional vs Transformer-Based Models\*\*

- Transformer-based models consistently outperform traditional machine learning models across all metrics. For example, the best traditional model, the MLP Neural Network, achieves a Test F1 score of 0.7059, which is notably lower than the lowest-performing transformer-based model, BERT-base, with a Test F1 score of 0.7401.

- Accuracy is highest for BioBERT (0.867) and DeBERTa (0.8655), surpassing the strongest traditional model, MLP Neural Network, which achieves 0.8186. Similarly, Precision and Recall are highest for DeBERTa (0.7602 and 0.7499, respectively), outperforming MLP Neural Network's Precision (0.7339) and Recall (0.6867).

##### 2. \*\*Best-Performing Model(s)\*\*

- DeBERTa is the best-performing model overall, achieving the highest Test F1 score (0.7546), Precision (0.7602), and Recall (0.7499). Although BioBERT slightly outperforms DeBERTa in Accuracy (0.867 vs. 0.8655), DeBERTa's balanced performance across all metrics makes it the optimal choice for clinical status prediction tasks.

##### 3. \*\*Trends, Weaknesses, and Trade-offs\*\*

- **\*\*Trends:\*\*** Transformer-based models demonstrate superior generalization capabilities, likely due to their pretraining on large-scale corpora and ability to capture contextual information. BioBERT's domain-specific pretraining on biomedical data provides a marginal advantage over generic BERT-base, evident from its improved Test F1 and Accuracy.

- **\*\*Weaknesses:\*\*** Traditional models, while computationally efficient, suffer from lower Recall, indicating potential underperformance in identifying minority classes, which may suggest class imbalance in the dataset. This weakness could exacerbate clinical risks where misclassification of critical statuses (e.g., severe conditions) is unacceptable.

- **\*\*Trade-offs:\*\*** Transformer-based models demand significant computational

resources for training and inference, which may limit their applicability in settings with constrained hardware. Additionally, their complexity reduces interpretability compared to models like Random Forest, which provide feature importance scores that are more accessible to clinicians.

#### 4. **\*\*Practicality for Clinical Deployment\*\***

- **\*\*Random Forest:\*\*** Its simplicity and interpretability make it suitable for resource-constrained settings; however, its lower performance limits its reliability for high-stakes clinical tasks.

- **\*\*XGBoost:\*\*** While offering improved performance over Random Forest, its complexity and susceptibility to hyperparameter tuning challenges may hinder usability in real-time clinical environments.

- **\*\*MLP Neural Network:\*\*** Despite higher accuracy than other traditional models, its lack of interpretability and computational demands reduce its practicality.

- **\*\*Transformer-Based Models:\*\*** While DeBERTa and BioBERT excel in predictive performance, their high computational costs may restrict deployment in low-resource clinics. However, their potential to improve decision-making in critical cases justifies investment in infrastructure upgrades.

#### 5. **\*\*Future Research Directions\*\***

- Addressing class imbalance and improving Recall for minority clinical statuses is essential to mitigate risks in misclassification. Techniques such as data augmentation, synthetic minority oversampling, or loss function adjustments (e.g., focal loss) should be explored.

#### 6. **\*\*Recommendation\*\***

- Based on the analysis, transformer-based models, particularly DeBERTa and BioBERT, are recommended for clinical applications requiring high predictive accuracy and reliability. However, deployment should be accompanied by infrastructure upgrades and interpretability tools to facilitate clinician trust and usability.