DSA4262 Individual Assignment 2 (Data Visualization + Predictive Modelling)

Introduction To The Project

Introducing Stress:

Stress has become an increasingly pervasive issue worldwide, affecting both mental and physical health. Studies indicate that a significant portion of the population experiences high stress levels regularly, which can contribute to conditions such as anxiety, depression and cardiovascular problems. With the rapid rise of social media, many users now openly discuss stressful incidents online, providing a rich source of data for understanding patterns of stress. Among various platforms, Reddit has emerged as a particularly informative space, where communities openly share personal experiences and challenges. The **Dreaddit project** leverages this growing body of user-generated content to study stress expressions on social media, aiming to systematically identify posts that reflect stressful experiences.

About The Dreaddit Project:

The Dreaddit project focuses on determining whether a given Reddit post can be classified as stressful or non-stressful. To achieve this, we incorporate multiple features derived from each post. These include raw text content, numerical metrics such as LIWC scores (capturing linguistic and psychological patterns), sentiment scores, confidence levels and DAL features. By combining these textual and numerical indicators, the project seeks to build a robust framework for accurately predicting the stress level of a post.

What The Project Is About:

To classify texts as stressful or non-stressful, we will develop a machine learning pipeline and evaluate its performance using F1-scores as the primary metric. The analysis will explore a wide range of models, including classical approaches like Logistic Regression and Support Vector Machines (SVM), ensemble methods such as Boosting models, as well as deep learning architectures including Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks. This diverse set of models allows us to compare performance across different techniques and identify the most effective approach for stress detection.

Datasets That We Will Be Using:

The datasets used in this study include `dreaddit_train`, which serves both training and validation purposes and `dreaddit_test`, which is reserved for evaluating the final model. The model achieving the highest F1-score on the validation set will likely be applied to the test dataset, and a feasibility analysis will be performed to ensure reliable performance in practice. This approach allows us to measure the generalizability of the model while maintaining rigorous evaluation standards.

Link To GitHub Repository:

Link:
https://github.com/tadamaen/DSA4262/tree/main/DSA4262%20Individual%20Assignment%

Importing Relevant Modules

1. **Pandas** - For Data Cleaning And Transformation
2. **Numpy** - For Data Manipulation
3. **Matplotlib** - For Data Visualization
4. **scikit-learn** - For Machine Learning
5. **Seaborn** - For Data Visualization
6. **PyTorch** - For Deep Learning

Note: I am using a different version of numpy. This is because torch only supports numpy 1.x versions and not numpy 2.x versions. Hence, I am using a numpy 1.x version instead. To run the code properly without any errors, please press "Restart Kernel And Run All Cells". Please ignore the error message that appears after the first block is run, the code works fine :)

In [4]:
```
!pip install "numpy==1.26.4" --upgrade --force-reinstall -qq
!pip install torch -qq
```

```
ERROR: pip's dependency resolver does not currently take into account all the pac
kages that are installed. This behaviour is the source of the following dependenc
y conflicts.
shap 0.50.0 requires numpy>=2, but you have numpy 1.26.4 which is incompatible.
streamlit 1.32.0 requires packaging<24,>=16.8, but you have packaging 26.0 which
is incompatible.
streamlit 1.32.0 requires pandas<3,>=1.3.0, but you have pandas 3.0.1 which is in
compatible.
```

In [5]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import torch
```

Importing The Given Datasets

There are two datasets that will used in this Predictive Modelling project. These include:

1. `dreaddit-train.csv`
2. `dreaddit-test.csv`

NOTE: These two files are in csv format and we will import the csv files directly using pandas `pd.read_csv` method.

Importing Dataset 1 - Dreaddit Train (CSV File)

```
In [7]: dreaddit_train = pd.read_csv('dreaddit-train.csv')
        dreaddit_train.head()
```

Out[7]:

	subreddit	post_id	sentence_range	text	id	label	confidence	soc
0	ptsd	8601tu	(15, 20)	He said he had not felt that way before, sugge...	33181	1	0.8	
1	assistance	8lbrx9	(0, 5)	Hey there r/assistance, Not sure if this is th...	2606	0	1.0	
2	ptsd	9ch1zh	(15, 20)	My mom then hit me with the newspaper and it s...	38816	1	0.8	
3	relationships	7rorpp	[5, 10]	until i met my new boyfriend, he is amazing, h...	239	1	0.6	
4	survivorsofabuse	9p2gbc	[0, 5]	October is Domestic Violence Awareness Month a...	1421	1	0.8	

5 rows × 116 columns

Importing Dataset 2 - Dreaddit Test (CSV File)

```
In [9]: dreaddit_test = pd.read_csv('dreaddit-test.csv')
        dreaddit_test.head()
```

Out[9]:

	id	subreddit	post_id	sentence_range	text	label	confidence	social_tim
0	896	relationships	7nu7as	[50, 55]	Its like that, if you want or not.“ ME: I have...	0	0.8	151
1	19059	anxiety	680i6d	(5, 10)	I man the front desk and my title is HR Custom...	0	1.0	149
2	7977	ptsd	8eeu1t	(5, 10)	We'd be saving so much money with this new hou...	1	1.0	152
3	1214	ptsd	8d28vu	[2, 7]	My ex used to shoot back with "Do you want me ...	1	0.5	152
4	1965	relationships	7r1e85	[23, 28]	I haven’t said anything to him yet because I’m...	0	0.8	151

5 rows × 116 columns

Data Cleaning

We will need to perform data cleaning for both the training and test sets. The data cleaning methods will include:

1. Removing unnecessary and irrelevant columns
2. Renaming columns to a more readable format
3. Standardizing the sentence range values (like removing inconsistencies [] vs (), number range)
4. Identification and handling of any missing values

Data Cleaning Process (Train + Test Sets)

Step 1: Removing Unnecessary Columns

Before proceeding with the visualizations, it is important to note that the Dreaddit training and test datasets contain a large number of features (116 columns). Many of these variables are not directly relevant to our analysis. Including an excessive number of features in subsequent predictive modelling may increase model complexity, raise the risk of overfitting and ultimately lead to poorer generalisation performance on the test set. Therefore, we perform a feature selection step to retain only the columns that are most relevant for exploratory analysis and model development.

A question that we should ask is which columns are considered important and which columns should we remove in order to reduce the number of columns as much as possible but still retain most of the important data for consideration? This is when we should explore each column one-by-one and determine whether the column is important.

Important Columns Include:

Category 1: Core Metadata And Annotation Columns

Column Name In Dataset	Revised Name	Why Important?
subreddit	Subcategory	Identifies the community context in which a post is written, enabling analysis of how stress expression varies across different social environments. It is essential for subgroup performance analysis and domain-specific interpretation
sentence_range	Sentence Range	Provides a coarse measure of post length and verbosity, which can reflect cognitive load or narrative stress. It is useful for analysing how stress correlates with the amount of text produced
text	Text	Contains the raw linguistic content of each post, which is necessary for qualittive inspection, error analysis, and feature extraction. It enables direct interpretation of model predictions and failure cases
label	Label	Represents the ground-truth stress annotation and serves as the target variable for supervised learning. It is required for both model training and performance evaluation

Column Name In Dataset	Revised Name	Why Important?
confidence	Confidence Level	Indicates the annotators confidence in the assigned label, reflecting annotation reliability. This feature is useful for analysing ambiguous cases and understanding disagreement in stress perception

Category 2: Sentiment And Social Context Features

Column Name In Dataset	Revised Name	Why Important?
sentiment	Sentiment	Captures the overall emotional polarity of a post, providing a high-level affective signal. It allows comparison between emotional tone and stress labels, highlighting cases where stress is expressed without overt negativity
social_upvote_ratio	Upvote Proportion	Measures the proportion of positive community feedback received by a post, serving as a proxy for social approval. It helps analyse how stressed versus non-stressed posts are received by the community
social_num_comments	Number Of Comments	Reflects the level of engagement a post generates, indicating social attention or concern. This feature provides contextual insight into how stress-related content elicits discussion

Category 3: LIWC Affective Features

Column Name In Dataset	Revised Name	Why Important?
lex_liwc_negemo	Negative Emotional Language	Measures the proportion of negative emotional language in a post. It is a strong and interpretable indicator of affective distress commonly associated with stress
lex_liwc_anx	Anxiety	Captures anxiety-related language, which is closely linked to psychological stress. This feature is particularly relevant for detecting anticipatory worry and tension

Column Name In Dataset	Revised Name	Why Important?
lex_liwc_sad	Sadness	Represents expressions of sadness, a frequent component of distress-related narratives. It helps distinguish emotionally heavy content from neutral discourse
lex_liwc_anger	Anger	Measures expressions of frustration and anger, which often co-occur with stress. It is useful for identifying stress manifested through irritability rather than sadness

Category 4: LIWC Cognitive Processing Features

Column Name In Dataset	Revised Name	Why Important?
lex_liwc_cogproc	Cognitive Processing	Captures cognitive processing language associated with thinking, reasoning, and mental effort. Stress is often linked to rumination and overthinking, making this feature highly informative
lex_liwc_insight	Introspective Language	Reflects introspective and self-reflective language. It is useful for identifying posts involving personal reflection, which often accompany stress narratives
lex_liwc_cause	Casual Reasoning Language	Measures causal reasoning language, indicating attempts to explain or rationalise situations. Such reasoning is common in stressed individuals trying to make sense of difficulties
lex_liwc_tentat	Tentative Language	Measures tentative language, signalling uncertainty and lack of confidence. Stress frequently manifests as hesitation or indecision in language use
lex_liwc_certain	Certainty	Represents expressions of certainty and conviction. It provides a useful contrast to tentative language, helping distinguish confident statements from uncertain, stress-related ones

Category 5: LIWC Self-Focus

Column Name In Dataset	Revised Name	Why Important?
lex_liwc_i	First Person Singular Pronouns	Captures personal pronoun usage more broadly. It reflects how much a post is centred on personal experience versus external topics
lex_liwc_we	First Person Plural Pronouns	This indicates collective focus or group orientation. High values suggest the writer is including themselves as part of a group.
lex_liwc_you	Second Person Pronouns	This indicates focus on the reader or another person. High values suggest the text is addressing someone directly.
lex_liwc_shehe	Third Person Singular Pronouns	This indicates focus on other individuals. High values suggest discussing someone else's perspective or actions.
lex_liwc_they	Third Person Plural Pronouns	This indicates focus on other groups. Useful for analyzing discussions about groups or communities.

Category 6: LIWC Social And Situational Stressors

Column Name In Dataset	Revised Name	Why Important?
lex_liwc_social	Social Interactions	Captures references to social interactions and relationships. Many stressors are interpersonal, making this feature valuable for contextual interpretation
lex_liwc_family	Family	Measures family-related language, which often reflects emotionally salient stress sources. It is useful for identifying relational and caregiving stress
lex_liwc_friend	Friends	Captures references to friendships and peer relationships. This feature supports analysis of social support and conflict in stress narratives

Column Name In Dataset	Revised Name	Why Important?
lex_liwc_work	Work	Measures work-related language, a common source of chronic stress. It is particularly relevant for subreddits focused on employment or career issues.
lex_liwc_money	Money	Captures financial language, reflecting economic pressure and insecurity. Financial stress is a major real-world stressor, making this feature highly interpretable.
lex_liwc_achieve	Achievement	Measures achievement-oriented language related to goals and performance. It helps identify stress arising from pressure to succeed or meet expectation
lex_liwc_risk	Risk	Captures language associated with danger and uncertainty. This feature is useful for identifying stress linked to perceived threats or instability

Category 7: LIWC Psychological Style And Social Expression

Column Name In Dataset	Revised Name	Why Important?
lex_liwc_Clout	Social Confidence	Captures degree of confidence, authority, or social dominance reflected in the language.
lex_liwc_Authentic	Authentic	Captures the extent to which language appears personal, honest, and self-revealing.
lex_liwc_Tone	Tone	Captures the overall emotional tone of the text (positive and negative emotional framing).

Category 8: DAL Semantic-Emotional Content

Column Name In Dataset	Revised Name	Why Important?
lex_dal_avg_activation	Emotional Activation	Captures the average level of emotional activation (calm to excited, passive to intense) expressed by words in the text.

Column Name In Dataset	Revised Name	Why Important?
lex_dal_avg_imagery	Mental Imagery	Captures the average vividness or concreteness of words (abstract thoughts vs. sensory, image-evoking language)
lex_dal_avg_pleasantness	Emotional Valence	Captures the average emotional positivity or negativity of words used

The other columns not listed in these categories are classified as not important or irrelevant for our visualization and modelling project, hence we will remove the other columns. We now have only **35 columns** instead of the initial 116 columns, while still keeping most of the important information that can be used for our predictive models.

In [12]:
```
# Extracting The Important Columns And Removing The Rest (Train Dataset)
dreaddit_train = dreaddit_train[['subreddit', 'sentence_range', 'text', 'label',
                                 'social_num_comments', 'lex_liwc_negemo', 'lex_
                                 'lex_liwc_cogproc', 'lex_liwc_insight', 'lex_li
                                 'lex_liwc_we', 'lex_liwc_you', 'lex_liwc_shehe'
                                 'lex_liwc_friend', 'lex_liwc_work', 'lex_liwc_m
                                 'lex_liwc_Authentic', 'lex_liwc_Tone', 'lex_dal

# Extracting The Important Columns And Removing The Rest (Test Dataset)
dreaddit_test = dreaddit_test[['subreddit', 'sentence_range', 'text', 'label', '
                                 'social_num_comments', 'lex_liwc_negemo', 'lex_
                                 'lex_liwc_cogproc', 'lex_liwc_insight', 'lex_li
                                 'lex_liwc_we', 'lex_liwc_you', 'lex_liwc_shehe'
                                 'lex_liwc_friend', 'lex_liwc_work', 'lex_liwc_m
                                 'lex_liwc_Authentic', 'lex_liwc_Tone', 'lex_dal
```

Step 2: Renaming Columns To A More Readable Format

Many of the columns names are still in unreadable formats like having underscores and names that do not make much sense. To keep all columns consistent, we will remove underscores and name all columns in camel case format.

In [14]:
```
# Renaming All Columns For Consistency And Understandability (Train Dataset)
dreaddit_train.rename(columns = {'subreddit': 'Subcategory', 'sentence_range': '
                                 'confidence': 'Confidence Level', 'sentiment':
                                 'social_num_comments': 'Number Of Comments', 'l
                                 'lex_liwc_anx': 'Anxiety', 'lex_liwc_sad': 'Sad
                                 'lex_liwc_cogproc': 'Cognitive Processing', 'le
                                 'lex_liwc_cause': 'Casual Reasoning Language',
                                 'lex_liwc_certain': 'Certainty', 'lex_liwc_i':
                                 'lex_liwc_we': 'First Person Plural Pronouns',
                                 'lex_liwc_shehe': 'Third Person Singular Pronou
                                 'lex_liwc_social': 'Social Interactions', 'lex_
                                 'lex_liwc_work': 'Work', 'lex_liwc_money': 'Mon
                                 'lex_liwc_Clout': 'Social Confidence', 'lex_liw
                                 'lex_dal_avg_activation': 'Emotional Activation
                                 'lex_dal_avg_pleasantness': 'Emotional Valence'

# Renaming All Columns For Consistency And Understandability (Test Dataset)
dreaddit_test.rename(columns = {'subreddit': 'Subcategory', 'sentence_range': 'S
                                'confidence': 'Confidence Level', 'sentiment': '
                                'social_num_comments': 'Number Of Comments', 'le
                                'lex_liwc_anx': 'Anxiety', 'lex_liwc_sad': 'Sadr
                                'lex_liwc_cogproc': 'Cognitive Processing', 'lex
                                'lex_liwc_cause': 'Casual Reasoning Language', '
                                'lex_liwc_certain': 'Certainty', 'lex_liwc_i': '
                                'lex_liwc_we': 'First Person Plural Pronouns', '
                                'lex_liwc_shehe': 'Third Person Singular Pronour
                                'lex_liwc_social': 'Social Interactions', 'lex_l
                                'lex_liwc_work': 'Work', 'lex_liwc_money': 'Mone
                                'lex_liwc_Clout': 'Social Confidence', 'lex_liwc
                                'lex_dal_avg_activation': 'Emotional Activation'
                                'lex_dal_avg_pleasantness': 'Emotional Valence'}
```

Step 3: Standardizing Sentence Range Column Values

We observe that the sentence range column values are not standardized and exist in different formats. Some ranges are enclosed in round brackets () while others are enclosed in square brackets []. Let's explore the distinct values of the Sentence Range column.

In [16]:
```
# Finding The Distinct Values Of Sentence Range Column
dreaddit_train['Sentence Range'].unique()
```

Out[16]:
```
<ArrowStringArray>
[  '(15, 20)',     '(0, 5)',    '[5, 10]',     '[0, 5]',   '(30, 35)',
   '[25, 30]',    '(5, 10)',   '(50, 55)',   '[15, 20]',   '[20, 25]',
 ...
 '(200, 205)',   '(14, 19)',   '[58, 63]',   '[46, 51]', '(235, 240)',
   '(67, 72)',   '(33, 38)',   '[51, 56]', '[170, 175]',   '[76, 81]']
Length: 173, dtype: str
```

We noticed that there are many different categories of sentence ranges. This might make it a problem for visualization as there will be too many categories for us to explore. We will standardize the sentence range column values to:

1. **Very Short Text**: 1 to 10 sentences
2. **Short Text**: 11 to 20 sentences
3. **Medium Text**: 21 to 30 sentences
4. **Long Text**: 30 to 50 sentences
5. **Very Long Text**: Over 50 sentences

To classify each text properly, we will take the two numbers and calcualate the mean of the two values. If the mean of the two values exists as a decimal, we will round the number up to the nearest integer. For example, if the sentence has a length range of (32, 37), we will first find the mean value, which is (32 + 37) / 2 = 34.5. The value of 34.5 is a decimal number and hence, we will round up the value to 35. Based on the classification above, a text of 35 sentences is under **Long Text**.

In [18]:
```
# Performing Standardization On Sentence Range Column Values
import math
import re

# Step 1: Function to extract numbers and compute the rounded-up mean sentence l
```

```
def compute_mean_sentence_length(range_str):
    numbers = list(map(int, re.findall(r'\d+', range_str)))
    mean_val = sum(numbers) / 2
    return math.ceil(mean_val)

# Step 2: Function to map mean length to categories
def categorize_text(mean_length):
    if 1 <= mean_length <= 10:
        return "Very Short Text"
    elif 11 <= mean_length <= 20:
        return "Short Text"
    elif 21 <= mean_length <= 30:
        return "Medium Text"
    elif 31 <= mean_length <= 50:
        return "Long Text"
    else:
        return "Very Long Text"

# Step 3: Apply to the dreaddit train dataset
dreaddit_train["Mean Sentence Length"] = (dreaddit_train["Sentence Range"].apply
dreaddit_train["Text Length"] = (dreaddit_train["Mean Sentence Length"].apply(ca
dreaddit_train.drop(columns=['Mean Sentence Length', 'Sentence Range'], inplace

# Step 4: Apply to the dreaddit test dataset
dreaddit_test["Mean Sentence Length"] = (dreaddit_test["Sentence Range"].apply(c
dreaddit_test["Text Length"] = (dreaddit_test["Mean Sentence Length"].apply(cate
dreaddit_test.drop(columns=['Mean Sentence Length', 'Sentence Range'], inplace =
```

Step 4: Handling Missing Values

Next, we will examine the dataset to identify any missing values across the columns. This step is essential to ensure data integrity and completeness. If missing entries are detected, appropriate strategies such as row removal or data imputation will be applied to address them.

In [20]:
```
# Check for missing values in dreaddit_train
print("Missing values in dreaddit_train:")
print(dreaddit_train.isnull().sum())

# Check for missing values in dreaddit_test
print("\nMissing values in dreaddit_test:")
print(dreaddit_test.isnull().sum())
```

```
Missing values in dreaddit_train:
Subcategory                        0
Text                               0
Label                              0
Confidence Level                   0
Sentiment                          0
Upvote Proportion                  0
Number Of Comments                 0
Negative Emotional Language        0
Anxiety                            0
Sadness                            0
Anger                              0
Cognitive Processing               0
Introspective Language             0
Casual Reasoning Language          0
Tentative Language                 0
Certainty                          0
First Person Singular Pronouns     0
First Person Plural Pronouns       0
Second Person Pronouns             0
Third Person Singular Pronouns     0
Third Person Plural Pronouns       0
Social Interactions                0
Family                             0
Friends                            0
Work                               0
Money                              0
Achievement                        0
Risk                               0
Social Confidence                  0
Authentic                          0
Tone                               0
Emotional Activation               0
Mental Imagery                     0
Emotional Valence                  0
Text Length                        0
dtype: int64

Missing values in dreaddit_test:
Subcategory                        0
Text                               0
Label                              0
Confidence Level                   0
Sentiment                          0
Upvote Proportion                  0
Number Of Comments                 0
Negative Emotional Language        0
Anxiety                            0
Sadness                            0
Anger                              0
Cognitive Processing               0
Introspective Language             0
Casual Reasoning Language          0
Tentative Language                 0
Certainty                          0
First Person Singular Pronouns     0
First Person Plural Pronouns       0
Second Person Pronouns             0
Third Person Singular Pronouns     0
Third Person Plural Pronouns       0
```

```
Social Interactions                  0
Family                               0
Friends                              0
Work                                 0
Money                                0
Achievement                          0
Risk                                 0
Social Confidence                    0
Authentic                            0
Tone                                 0
Emotional Activation                 0
Mental Imagery                       0
Emotional Valence                    0
Text Length                          0
dtype: int64
```

There are no missing values in both the `dreaddit_train` and `dreaddit_test` datasets, hence we do not need to apply any imputation and removal of any rows in the datasets. For now, the dreaddit training and test dataset has been cleaned and we can now proceed to exploratory data analysis.

NOTE: For the visualizations, we will only be using the **training set** and not the test set. (The only exception is the introductory visualization where we analyze the distribution of stressful vs non-stressful texts in order to check if the dataset is balanced or imbalanced)

Exploratory Data Analysis - Data Visualization

We now proceed with a series of data visualizations to further explore the dataset and uncover meaningful patterns. Each visualization is accompanied by analytical commentary that offers possible explanations for the observed trends and anomalies.

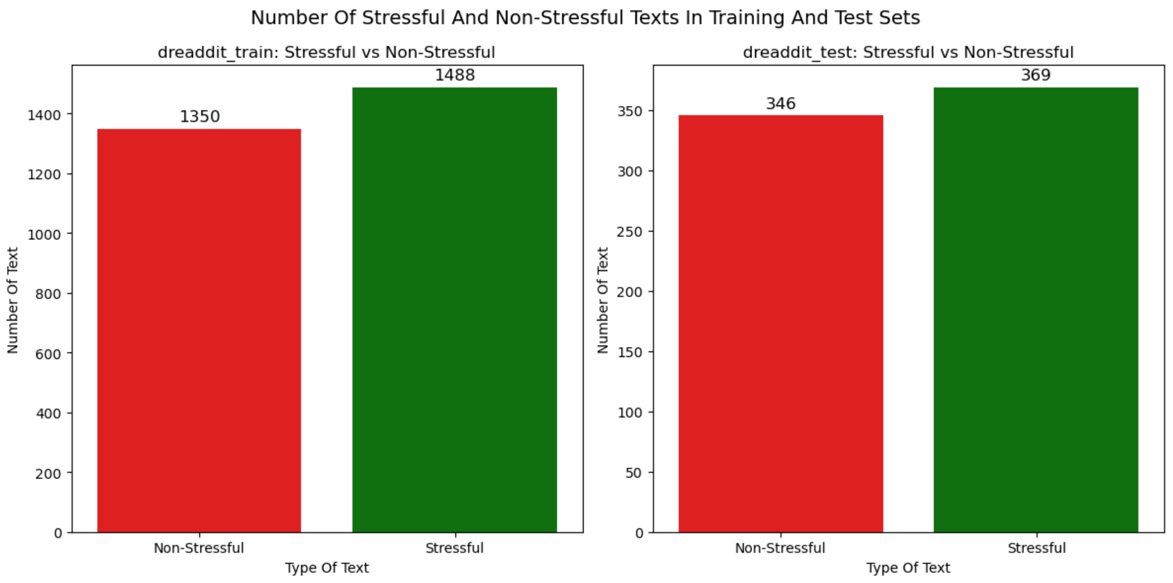
For the enlarged versions of the visualizations, please refer to this link: https://github.com/tadamaen/DSA4262/tree/main/DSA4262%20Individual%20Assignment%

Introductory Visualization: Distribution Of Stressful VS Non-Stressful Texts In Training And Testing Sets

To better understand the composition of our dataset, we analyze the distribution of stressful versus non-stressful texts in both the training and testing sets. This comparison helps us identify whether the labels are balanced or skewed, which is critical for building reliable predictive models. If the data is highly imbalanced, the model may become biased toward the majority class, reducing its ability to correctly classify minority cases. In such situations, data rebalancing techniques such as undersampling, oversampling or synthetic data generation methods like SMOTE can be applied to improve model performance and ensure fair representation of both classes during training.

In [22]:
```python
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Define mapping for labels
label_map = {0: "Non-Stressful", 1: "Stressful"}
```

```
palette = {"Non-Stressful": "red", "Stressful": "green"}
fig.suptitle("Number Of Stressful And Non-Stressful Texts In Training And Test S

# Training set
train_counts = dreaddit_train['Label'].map(label_map).value_counts().reindex(["N
sns.barplot(x=train_counts.index, y=train_counts.values,
            hue=train_counts.index, dodge=False,
            palette=palette, legend=False, ax=axes[0])
axes[0].set_title("dreaddit_train: Stressful vs Non-Stressful")
axes[0].set_xlabel("Type Of Text")
axes[0].set_ylabel("Number Of Text")
for i, val in enumerate(train_counts.values):
    axes[0].text(i, val + (val * 0.01), str(val), ha='center', va='bottom', font

# Testing set
test_counts = dreaddit_test['Label'].map(label_map).value_counts().reindex(["Nor
sns.barplot(x=test_counts.index, y=test_counts.values,
            hue=test_counts.index, dodge=False,
            palette=palette, legend=False, ax=axes[1])
axes[1].set_title("dreaddit_test: Stressful vs Non-Stressful")
axes[1].set_xlabel("Type Of Text")
axes[1].set_ylabel("Number Of Text")
for i, val in enumerate(test_counts.values):
    axes[1].text(i, val + (val * 0.01), str(val), ha='center', va='bottom', font

plt.tight_layout()
plt.show()
```



Dataset Distribution Insights:

Both the training and testing sets contain a marginally higher number of stressful texts compared to non-stressful ones.

- Train set: 1488 stressful vs. 1350 non-stressful.
- Test set: 369 stressful vs. 346 non-stressful.

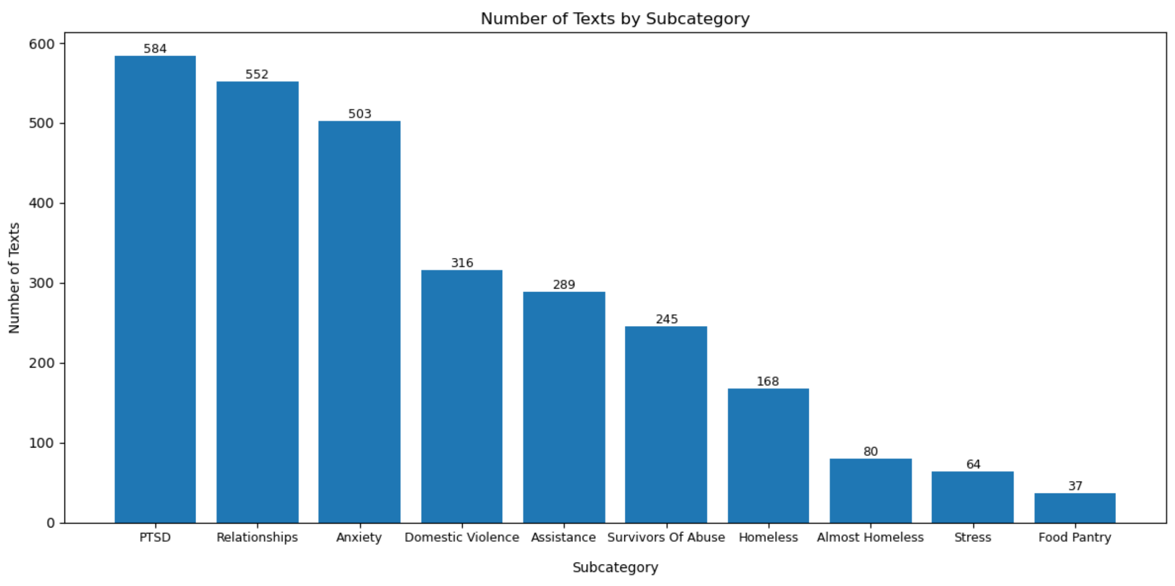
The imbalance is present, though it is not severe. However, even small imbalances can influence predictive models, especially in classification tasks where minority classes may be underrepresented in learned decision boundaries. To ensure fairness and robustness, we will factor this imbalance into consideration during the predictive modeling stage.

Visualization 1: Distribution Of Positive And Negative Stress Labels By Sub-Category

This visualization examines the relationship between **Subcategory** (eg. PTSD, assistance, relationships) and **Label** (0 negative, 1 = positive indicator of stress). We first begin by computing the total number of texts within each subcategory to understand the overall distribution of data across categories. Next, texts within each subcategory are disaggregated by their labels - positive (label = 1) and negative (label = 0). To enable fairer comparisons across subcategories with differing sample sizes, the absolute counts are converted into percentages. This normalization ensures that subcategories with larger volumes of texts do not disproportionately influence the interpretation relative to smaller subcategories.

Visualization 1 Part 1: Total Number Of Texts Within Each Subcategory

```
In [24]: # Count number of texts per subcategory
subcategory_counts = dreaddit_train["Subcategory"].value_counts().sort_values(as

# Rename subcategory labels
subcategory_rename_map = {"ptsd": "PTSD", "assistance": "Assistance", "relations
                          "survivorsofabuse": "Survivors Of Abuse", "domesticvic
                          "anxiety": "Anxiety", "homeless": "Homeless", "stress"
                          "almosthomeless": "Almost Homeless", "food_pantry": "F

subcategory_counts.index = subcategory_counts.index.map(subcategory_rename_map)

# Plot bar chart
plt.figure(figsize=(12, 6))
bars = plt.bar(subcategory_counts.index, subcategory_counts.values)

plt.title("Number of Texts by Subcategory")
plt.xlabel("Subcategory", labelpad = 12)
plt.ylabel("Number of Texts")
plt.xticks(fontsize=9)

# Add value labels above each bar
for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width() / 2, height, f"{int(height)}",
             ha="center", va="bottom", fontsize=9)

plt.tight_layout()
plt.show()
```



Key Observations:

1. **PTSD** (584 texts), **Relationships** (552 texts) and **Anxiety** (503 texts) dominate the dataset. These three categories together account for nearly 60% of all texts in the training dataset, showing where most of the Reddit users concerns cluster.

2. **Domestic Violence** (316 texts), **Assistance** (289 texts) and **Survivors Of Abuse** (245 texts) form a second tier. These represent serious social and safety issues, but with fewer texts compared to mental health concerns.

3. Lower Frequency Categories include **Homeless** (168 texts), **Almost Homeless** (80 texts), **Stress** (64 texts) and **Food Pantry** (37 texts). They have much smaller number of texts and these categories highlight material needs and situational crisis. However, they are underrepresented in the dataset.

Possible Insights:

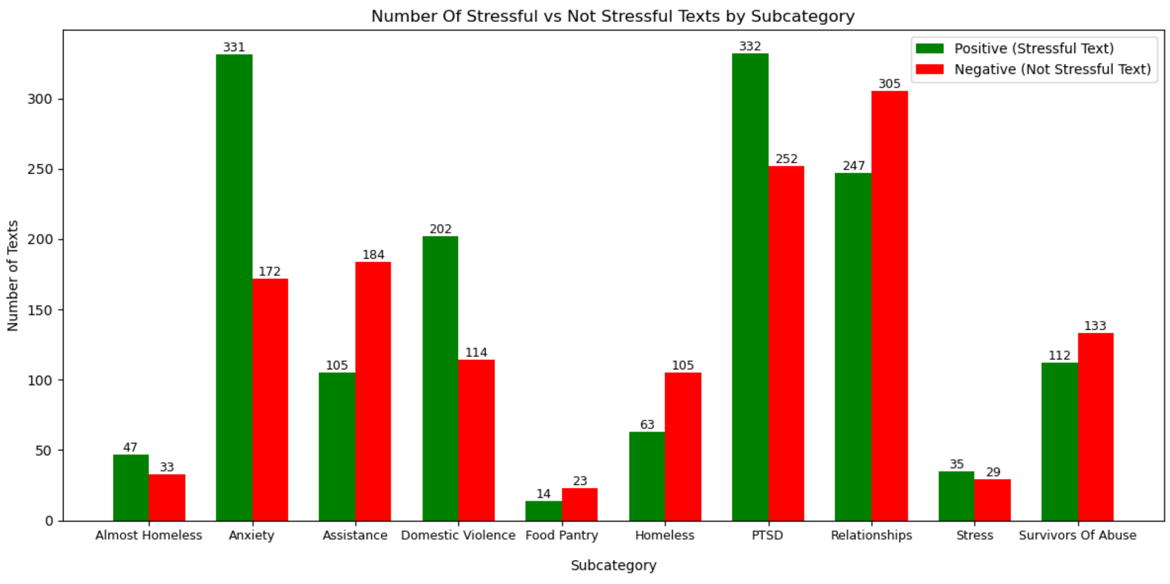
The dataset is heavily skewed toward psychological struggles (PTSD, Anxiety, Relationships) rather than material hardships (Homelessness, Food Pantry). This suggests Dreaddit users are more lilely to seek support for emotional and relational issues than for logistical or survival needs. The prominance of Relationships and Anxiety reflects the Draeddit community's role as a space for interpersonal and emotional support. Meanwhile, the smaller counts for Homelessness and Food Pantry suggest fewer users turn to Reddit for material aid compared to emotional support.

Visualization 1 Part 2: Number Of Positive And Negative Label Texts By Subcategory

The initial plot provides an overview of how texts are distributed across subcategories, but it does not convery any information about the nature or severity of stress expressed within those texts. A subcategory with a large number of posts may not necessarily indicate a higher prevalence of depressive indicators, it may simply reflect higher engagement or posting activity within that topic.

In the next visualization, the texts within each subcategory are further disagregated by their labels - positive (label = 1) and negative (label = 0) and the frequency of each label type is calculated. This deeper breakdown allows us to distinguish between subcategories that predominantly contain expressions of stress and those that do not, providing more meaningful insights into the emotional content of each subcategory rather than relying solely on volume-based comparisons.

In [26]:
```
# Count positive and negative labels per subcategory
label_counts = dreaddit_train.groupby('Subcategory')['Label'].value_counts().uns

# Rename subcategory labels
subcategory_rename_map = {"ptsd": "PTSD", "assistance": "Assistance", "relations
                          "domesticviolence": "Domestic Violence", "anxiety": "A
                          "almosthomeless": "Almost Homeless", "food_pantry": "F
label_counts.index = label_counts.index.map(subcategory_rename_map)

# Prepare bar positions for grouped bar chart
subcategories = label_counts.index
x = np.arange(len(subcategories))
width = 0.35

# Plot grouped bar chart
fig, ax = plt.subplots(figsize=(12, 6))
bars_positive = ax.bar(x - width/2, label_counts.get(1, 0), width, label='Positi
bars_negative = ax.bar(x + width/2, label_counts.get(0, 0), width, label='Negati

# Add labels, titles, and ticks
ax.set_xlabel('Subcategory', labelpad=12)
ax.set_ylabel('Number of Texts')
ax.set_title('Number Of Stressful vs Not Stressful Texts by Subcategory')
ax.set_xticks(x)
ax.set_xticklabels(subcategories, fontsize=9)
ax.legend()

# Add value labels above each bar
def add_labels(bars):
    for bar in bars:
        height = bar.get_height()
        ax.text(bar.get_x() + bar.get_width()/2, height, f'{int(height)}',
                ha='center', va='bottom', fontsize=9)

add_labels(bars_positive)
add_labels(bars_negative)
plt.tight_layout()
plt.show()
```



Key Observations:

1. **High Stress Categories** - This includes **Anxiety** (331 stressful vs 172 non-stressful), **PTSD** (332 stressful vs 252 non-stressful) and **Domestic Violence** (202 stressful vs 114 non-stressul). These categories are strongly associated with emotional distress, aligning with clinical and social severity.

2. **Low Stress Categories** - **Relationships** (247 stressful vs 305 non-stressful) has more non-stressful texts, suggesting that not all relationship-related posts are framed as crises - many may be just seeking advice. **Assistance** (105 stressful vs 184 non-stressful) also leans toward non-stressful, indicating that requests for help may often be practical rather than emotionally charged.

3. **Balanced Categories** - **Stress** (35 stressful vs 29 non-stressful) and **Survivors Of Abuse** (112 stressful vs 133 non-stressful) are relatively balanced, showing that these topics can generate both stressful and non-stressful narratives depending on context.

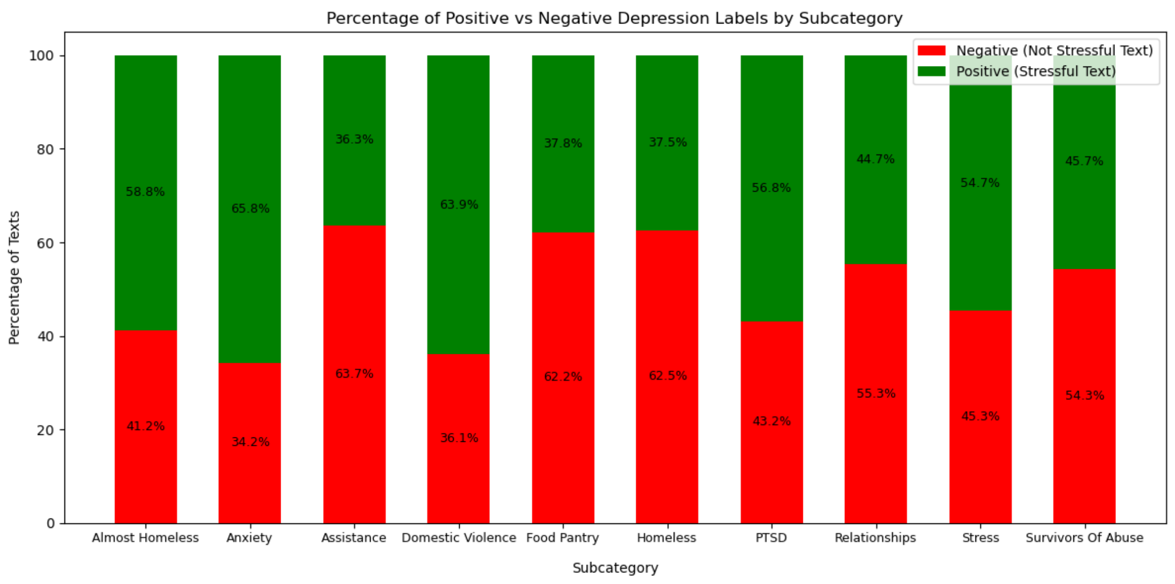
Visualization 1 Part 3: Percentage Of Positive And Negative Label Texts By Subcategory

The previous visualization provides a useful breakdown of positive and negative labels within each subcategory, but using absolute counts alone can be misleading when comparing subcategories of vastly different sizes. For example, PTSD subcategory may have hundreds of texts while Food Pantry may only have a few dozen. Even if both subcategories have a similar proportion of positive labels, the raw counts would suggest that PTSD is overwhelmingly more "stressful" than Food Pantry, which is not necessarily true.

To address this imbalance, the next visualization is employed to convert the counts into percentages relative to the total number of texts in each subcategory. By using percentages, we can accurately interpret trends across subcategories regardless of their sample size, ensuring that similar subcategories are not overshadowed by larger ones.

In [28]:

```python
# Calculate counts and convert to percentages
label_counts = dreaddit_train.groupby('Subcategory')['Label'].value_counts().uns

# Rename subcategories
subcategory_rename_map = {"ptsd": "PTSD", "assistance": "Assistance", "relations
                          "domesticviolence": "Domestic Violence", "anxiety": "A
                          "almosthomeless": "Almost Homeless", "food_pantry": "F
label_counts.index = label_counts.index.map(subcategory_rename_map)

# Convert counts to percentages per subcategory
label_percent = label_counts.div(label_counts.sum(axis=1), axis=0) * 100

# Plot stacked bar chart (percentage)
subcategories = label_percent.index
x = np.arange(len(subcategories))
width = 0.6
fig, ax = plt.subplots(figsize=(12, 6))
bars_negative = ax.bar(x, label_percent.get(0, 0), width, color='red', label='Ne
bars_positive = ax.bar(x, label_percent.get(1, 0), width, bottom=label_percent.g

# Add value labels inside each bar
for i in range(len(subcategories)):
    neg_height = label_percent.get(0, 0).iloc[i]
    pos_height = label_percent.get(1, 0).iloc[i]
    if neg_height > 0:
        ax.text(x[i], neg_height/2, f'{neg_height:.1f}%', ha='center', va='cente
    if pos_height > 0:
        ax.text(x[i], neg_height + pos_height/2, f'{pos_height:.1f}%', ha='cente

# Customize axes, title, legend
ax.set_xlabel('Subcategory', labelpad=12)
ax.set_ylabel('Percentage of Texts')
ax.set_title('Percentage of Positive vs Negative Depression Labels by Subcategor
ax.set_xticks(x)
ax.set_xticklabels(subcategories, fontsize=9)
ax.legend()
plt.tight_layout()
plt.show()
```



Key Observations:

1. **High-Stress Categories** (Majority Stressful)

- **Anxiety**: 65.8% stressful, 34.2% not stressful
- **Domestic Violence**: 63.9% stressful, 36.1% not stressful
- **Almost Homeless**: 58.8% stressful, 41.2% not stressful
- **PTSD**: 56.8% stressful, 43.2% not stressful

2. **Low-Stress Categories** (Majority Not Stressful)

- **Assistance**: 63.7% non-stressful, 36.3% stressful
- **Food Pantry**: 62.2% non-stressful, 37.8% stressful
- **Homeless**: 62.5% non-stressful, 37.5% stressful

3. **Balanced/Mixed Categories**

- **Relationships**: 55.3% non-stressful, 44.7% stressful
- **Survivors Of Abuse**: 54.3% non-stressful vs 45.7% stressful (These sit near the middle, showing that experiences in these areas can be noth stressful and reflective/advisory)

Possible Insights:

Although in raw counts, **PTSD** and **Anxiety** looked dominant, but percentages show that **Domestic Violence** and **Almost Homeless** are equally or more skewed toward stressful texts. This highlights categories where distress is proportionally higher even if total volume is lower. **Relationships** is a complex category. Despite being one of the largest categories, relationships still tilts slightly towards non-stressful (55.3%), showing that not all interpersonal posts are crisis-driven, many may be just seeking advice or reflective ones.

Visualization 2: Effect Of Text Length On Stress Labels

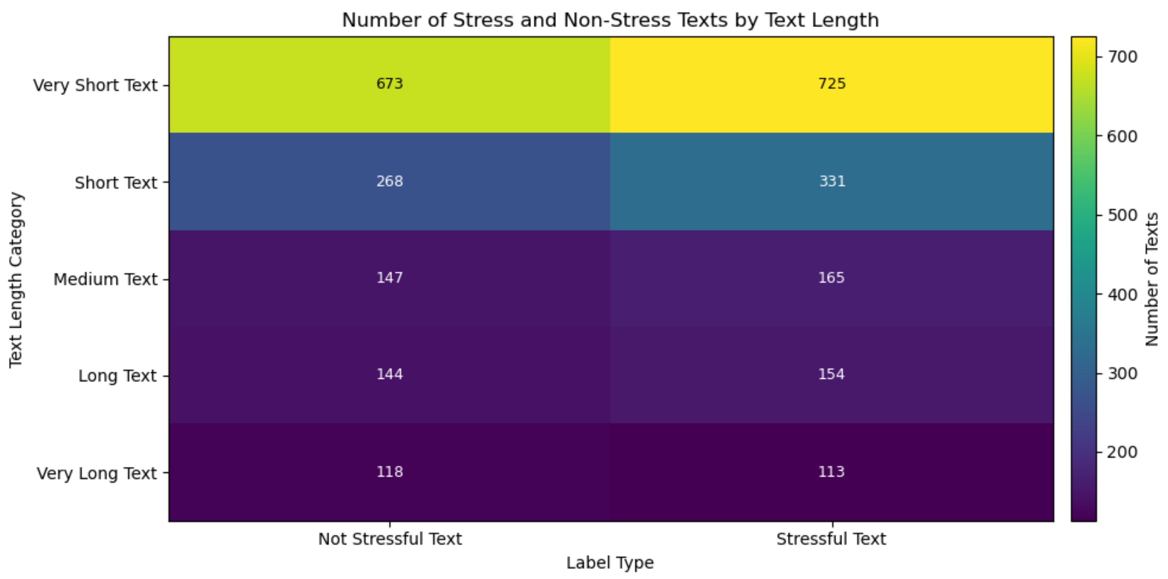
To further understand how textual characteristics relate to stress indicators, we examine the relationship between **text length categories** (Very Short Text, Short Text, Medium Text, Long Text, Very Long Text) and the **assigned labels** (0 = Not Stressful Text, 1 = Stressful Text). First, we compute the absolute number of texts within each text length category that are labelled as positive and negative. This provides an initial sense of how label distributions very across different text lengths.

However, because some text length categories may contain substantially more texts than others, absolute counts alone may led to biased interpretation. Therefore, we will additionally compute the absolute percentages of positive and negative labels within each text length category. The percentage-based perspective allows for a fairer comparison across text lengths, highlighting whether longer or shorter texts are proportionally more likely to express stressful indicators.

Visualization 2 Part 1: Number Of Texts With Positive And Negative Labels By Text Length

To examine how stress labels vary across different text length categories, a **heatmap** is used to visualize the relationship between text length and label type. The heatmap displays the absolute number of texts within each text length category that are labeled as positive or negative. Color intensity represents magnitude, making it easy to identify patterns. A Viridis color scale is employed to ensure perceptual uniformity and accessibility.

In [30]:
```
# Create count-based contingency table
heatmap_counts = pd.crosstab(dreaddit_train["Text Length"], dreaddit_train["Labe
heatmap_counts.columns = ["Negative", "Positive"]
ordered_lengths = ["Very Short Text", "Short Text", "Medium Text", "Long Text",
heatmap_counts = heatmap_counts.loc[ordered_lengths]

# Plot heatmap
fig, ax = plt.subplots(figsize=(10, 5))
im = ax.imshow(heatmap_counts.values, cmap="viridis", aspect="auto")
ax.set_xticks(np.arange(len(heatmap_counts.columns)))
ax.set_yticks(np.arange(len(heatmap_counts.index)))
ax.set_xticklabels(["Not Stressful Text", "Stressful Text"])
ax.set_yticklabels(heatmap_counts.index)
ax.set_xlabel("Label Type")
ax.set_ylabel("Text Length Category")
ax.set_title("Number of Stress and Non-Stress Texts by Text Length")

threshold = 500
for i in range(len(heatmap_counts.index)):
    for j in range(len(heatmap_counts.columns)):
        value = heatmap_counts.iloc[i, j]
        text_color = "black" if value >= threshold else "white"
        ax.text(j, i, value, ha="center", va="center", color=text_color, fontsiz

cbar = plt.colorbar(im, ax=ax, fraction=0.035, pad=0.02)
cbar.set_label("Number of Texts")
plt.tight_layout()
plt.show()
```



Key Observations:

1. **Very Short Text** (1 to 10 sentences) dominate. There are **673 non-stressful** vs **725 stressful** texts, which represents tha largest category overall. This suggests that

many users express distress or support in brief, concise posts rather than long narratives.

2. **Short Text** (11 to 20 sentences) are the second largest, with **268 non-stressful** and **331 stressful texts**, still skewed towards stressful. This indicates that short posts often carry emotional weight, possibly quick disclosures or urgent statements.

3. For the other categories like **Medium Text** (21 to 30 sentences), **Long Text** (31 to 50 sentences) and **Very Long Text** (Over 50 sentences), the number of non-stressful and stressful texts are more balanced.

Possible Insights:

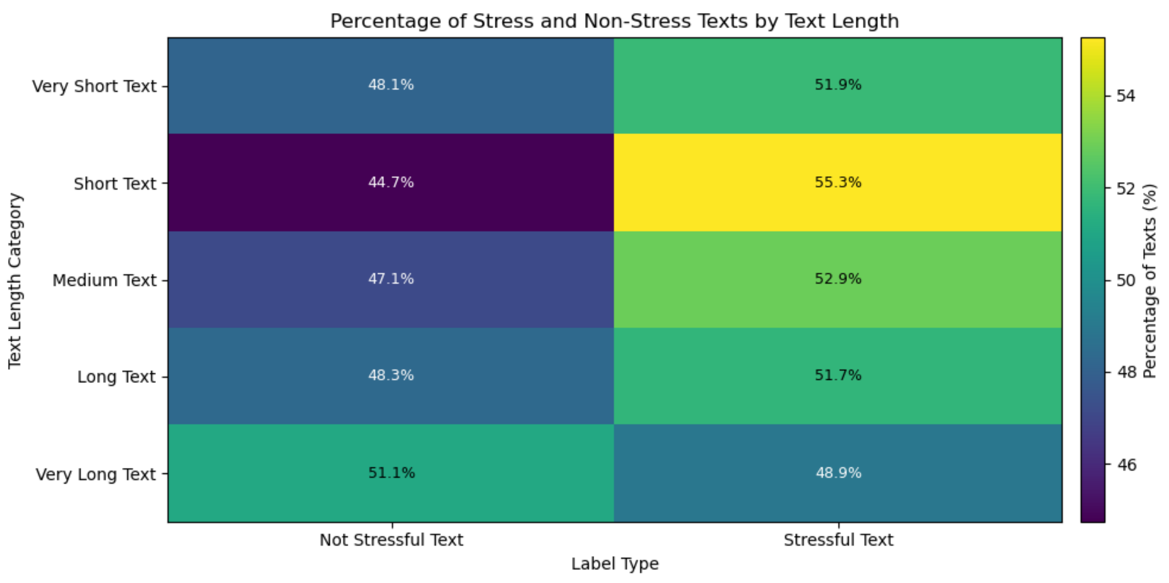
1. **Stress Is Expressed Concisely** - The majority of stressful texts are short or very short showing that distress often comes in brief bursts of communication. This aligns with the idea that people in crisis may not elaborate but instead post quickly and urgently.

2. **Longer Texts Are More Reflective In Nature** - As text length increases, the balance between stressful and non-stressful posts evens out. Longer posts may include context, reflections or advice-seeking, diluting the overall stress signal.

Visualization 2 Part 2: Percentage Of Texts With Positive And Negative Labels By Text Length

The percentage-based heatmap illustrates the proportion of texts within each text length category that are labeled as stress or not stress texts, providing insight into how the likelihood of expressing stress-related content varies with text length. By examining percentages rather than raw counts, the visualization reveals patterns that are not driven by the volume of texts in each category but by their relative composition. For instance, even if very short texts appear frequently in the dataset, the heatmap allows us to assess whether they are proportionally more or less likely to indicate stress compared to longer texts.

In [32]:
```python
heatmap_percent = (pd.crosstab(dreaddit_train["Text Length"],
                               dreaddit_train["Label"], normalize="index") * 100
heatmap_percent.columns = ["Not Stressful Text", "Stressful Text"]
ordered_lengths = ["Very Short Text", "Short Text", "Medium Text", "Long Text",
heatmap_percent = heatmap_percent.loc[ordered_lengths]

# Plot heatmap
fig, ax = plt.subplots(figsize=(10, 5))
im = ax.imshow(heatmap_percent.values, cmap="viridis", aspect="auto")
ax.set_xticks(np.arange(len(heatmap_percent.columns)))
ax.set_yticks(np.arange(len(heatmap_percent.index)))
ax.set_xticklabels(heatmap_percent.columns)
ax.set_yticklabels(heatmap_percent.index)
ax.set_xlabel("Label Type")
ax.set_ylabel("Text Length Category")
ax.set_title("Percentage of Stress and Non-Stress Texts by Text Length")

for i in range(len(heatmap_percent.index)):
    for j in range(len(heatmap_percent.columns)):
        value = heatmap_percent.iloc[i, j]
        ax.text(j, i, f"{value:.1f}%", ha="center", va="center",
                color="black" if value >= 50 else "white", fontsize=9)

cbar = plt.colorbar(im, ax=ax, fraction=0.035, pad=0.02)
cbar.set_label("Percentage of Texts (%)")
plt.tight_layout()
plt.show()
```



Key Observations:

1. **Short Text** (10 to 20 sentences) show the highest proportion of stressful posts (55.3%) while **Very Short Text** (1 to 10 sentences) and **Medium Text** (21 to 30 sentences) also skew slightly towards stressful at 51.9% and 52.9% respectively.

2. **Long Text** (31 to 50 sentences) remain fairly balanced, with 51.7% stressful and 48.3% non-stressful. Interestingly, **Very Long Text** (Over 50 sentences) is the only category where non-stressful posts slightly outweigh stressful ones, at 51.1% compared to 48.9%. This suggests that while stress is more commonly expressed in shorter posts, longer narratives tend to balance out or even shift toward non-stressful framing.

Possible Insights:

Distress is often communicated in brief, urgent statements rather than lengthy explanations. Short texts are more likely to carry stressful content, which aligns with the ideas that indicidicials in crisis may not elaborate but instead post quickly and directly. Longer texts appear to dilute stress signals as they often include context, reflection and advice-seeking, causing more non-stressful framing to be introduced to the text.

A possible reason why **Very Short Text** category (1 to 10 sentences) have lower than expected proportion of stressful text is that extremely short messages often lack the emotional detail needed to be labelled as stressful. Many of these posts may be neutral stetements, clarifications or simple requests that do not explicitly convey distress. In contrast, **Short Text** (11 to 20 sentences) provide just enough context or emotional language to signal distress more clearly.

Visualization 3: Identifying Most Common Words Used In Stressful And Non Stressful Texts

To better understand the linguistic differences between stressful and non-stressful texts, we analyze the most frequently used words associated with each label. For each label, all texts are first combined into a single corpus, allowing us to capture overall word usage patterns within each category. Common words that do not contribute meaningful semantic information such as articles and conjunctions are then removed using a predefined list of stopwords. This preprocessing step ensures that the resulting visualizations highlight content-bearing words that better reflect emotional tone and thematic emphasis.

Two complementary visualizations are used. First, a word cloud provides an intuitive, high-level overview of commonly used words, where word size corresponds to frequency. Second, we plot the top 10 most frequently used words for each label using bar charts, where the x-axis represents words and the y-axis represents their frequency. Together, these visualizations allow us to identify both prominent themes and measurable differences in language usage between stressful and non-stressful texts.

Visualization 3 Part 1: Word Clouds Of Words Used In Stressful And Not Stressful Texts

In [34]:
```
!pip install wordcloud -q
from wordcloud import WordCloud
from collections import Counter
import re
import nltk
from nltk.corpus import stopwords

# Download stopwords
nltk.download("stopwords")
```

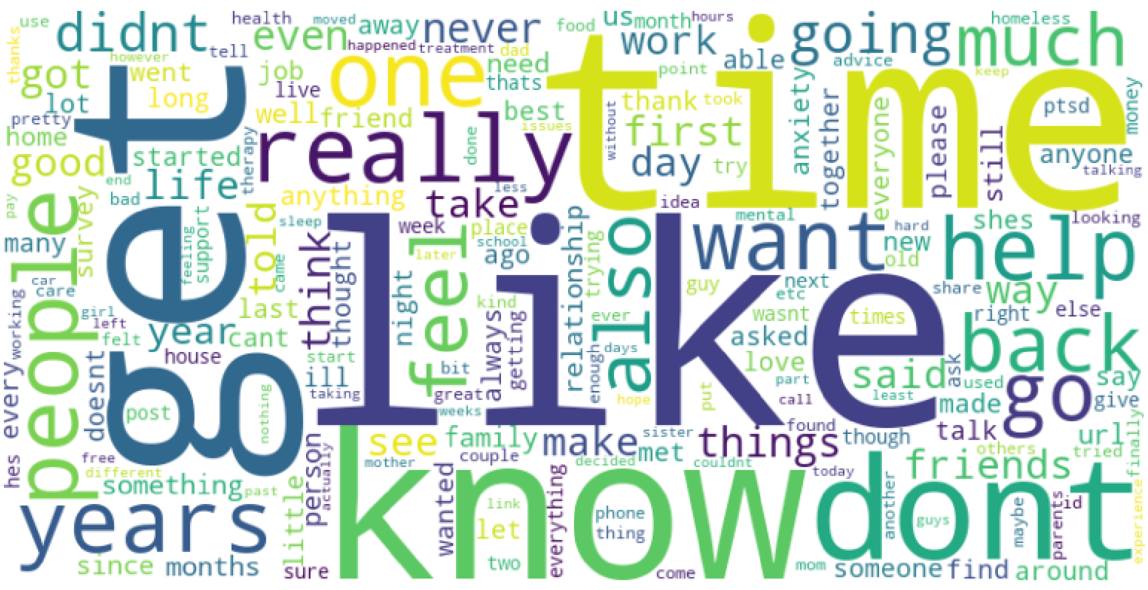
```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\tadam\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[34]:
```
True
```

In [35]:
```
# Text preprocessing function
custom_stopwords = set(stopwords.words("english"))
extra_stopwords = {"im", "ive", "would", "could", "should", "may", "might", "mus
custom_stopwords = custom_stopwords.union(extra_stopwords)

def preprocess_text(text):
    text = text.lower()
    text = re.sub(r"[^a-z\s]", "", text)
    tokens = text.split()
    tokens = [word for word in tokens if word not in custom_stopwords]
    return tokens

# Combine texts by label
stress_text = " ".join(dreaddit_train[dreaddit_train["Label"] == 1]["Text"])
non_stress_text = " ".join(dreaddit_train[dreaddit_train["Label"] == 0]["Text"])

# Preprocessing texts
```

```
stress_tokens = preprocess_text(stress_text)
non_stress_tokens = preprocess_text(non_stress_text)

# Generate word frequencies
stress_word_freq = Counter(stress_tokens)
non_stress_word_freq = Counter(non_stress_tokens)

# Plotting Word Cloud For Non-Stressful Texts
wordcloud_non_stress = WordCloud(width=800, height=400, background_color="white"
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud_non_stress, interpolation="bilinear")
plt.title("Word Cloud: Non-Stressful Texts")
plt.axis("off")
plt.show()
```

Word Cloud: Non-Stressful Texts

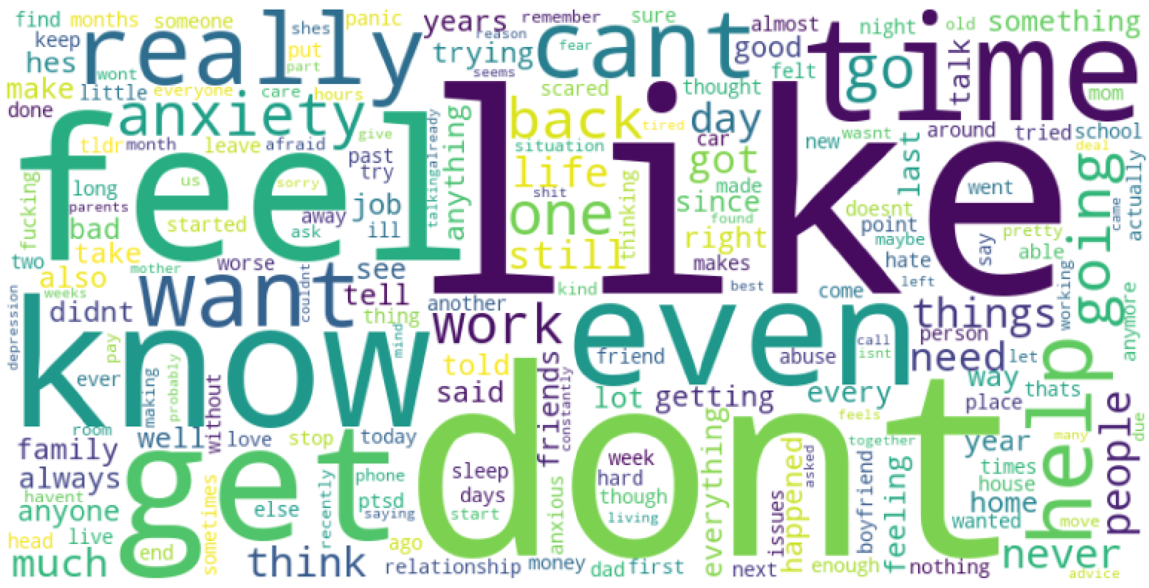


The most prominent words in the word cloud include **time, know, like, back, years, people, help, said, really, want, good, friends, life, feel, going, one and take**. These words are larger, indicating higher frequency in non-stressful texts. Many of them are neutral or positive in tone - they suggest supportive or reflective contexts. Other words point to temporal framing, where users are recounting experiences or reflecting on the past.

The vocabulary is dominated by general conversational words rather than highly emotional or crisis-driven terms. Most words reflect everyday language patterns, suggesting that non-stressful texts often resemble casual conversations or reflective storytelling rather than urgent disclosures.

In [37]:
```
# Plotting Word Cloud For Stressful Texts
wordcloud_stress = WordCloud(width=800, height=400, background_color="white", cc
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud_stress, interpolation="bilinear")
plt.title("Word Cloud: Stressful Texts")
plt.axis("off")
plt.show()
```

Word Cloud: Stressful Texts



In the stressful texts word cloud, the most prominent words include **like, dont, know, feel, get, time, even, want, anxiety, really, back, still, help, people, life, work and things.** Compared to the non-stressful word cloud, there is a noticeable shift in tone and vocabulary. Stressful texts feature more negative or uncertain expressions which directly convey emotional struggle. Several words also suggest practical burdens or frustrations.

Possible Insights:

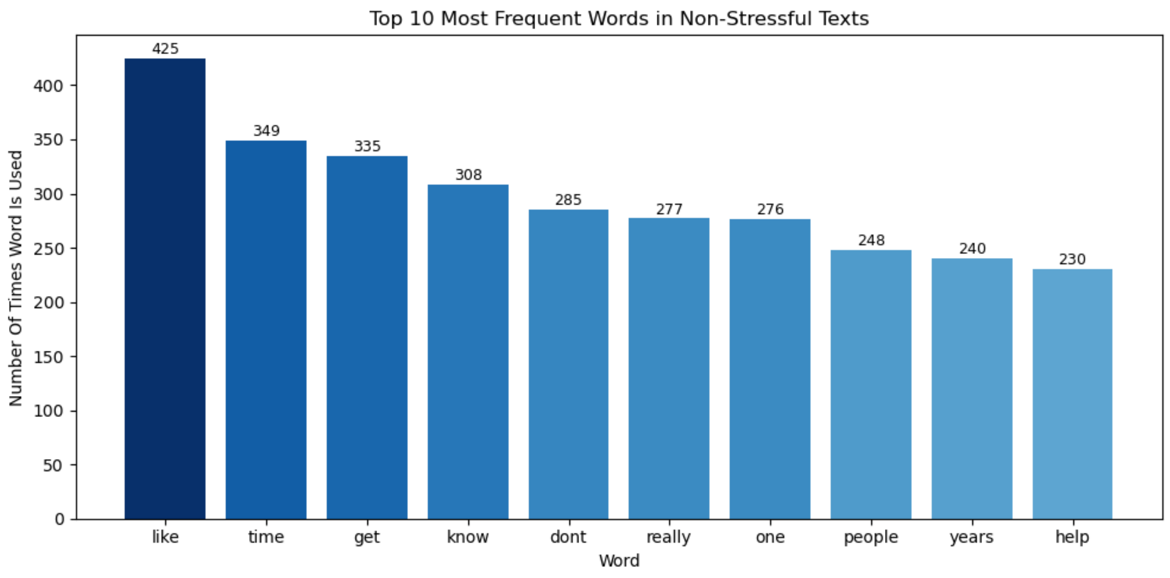
1. The prevalence of words such as friends, help and good in non-stressful texts suggests that non-stressful texts often revolve around social support, positive experiences or advice-seeking. These posts may be more about sharing life updates, seeking guidance or expressing gratitude rather than conveying distress. The frequent use of temporal markers like time, years and back indicates that non-stressful texts often involve reflection, where users recount past events in a calmer, less emotionally charged way.

2. Another insight is that while both stressful and non-stressful texts use common conversational words, the co-text and framing differ. In stressful texts, these words are embedded in narratives of struggle whereas in non-stressful texts, they appear in more neutral or positive exchanges. This highlights how linguistic overlap masks divergent emotional tones, making context crucial for classification.

Visualization 3 Part 2: Plotting 10 Most Frequently Used Words In Stressful And Non Stressful Texts

In [39]:
```
# Top 10 most frequent words
top10_stress = stress_word_freq.most_common(10)
top10_non_stress = non_stress_word_freq.most_common(10)
df_stress = pd.DataFrame(top10_stress, columns=["Word", "Count"])
df_non_stress = pd.DataFrame(top10_non_stress, columns=["Word", "Count"])

# Plotting Top 10 most frequent words for Non-Stressful Texts

# Normalize counts for color intensity (shades of blue)
counts = df_non_stress["Count"].values
```

```
colors = plt.cm.Blues(counts / counts.max())

# Plot bar chart
plt.figure(figsize=(10, 5))
bars = plt.bar(df_non_stress["Word"], counts, color=colors)
plt.title("Top 10 Most Frequent Words in Non-Stressful Texts")
plt.xlabel("Word")
plt.ylabel("Number Of Times Word Is Used")
for bar, count in zip(bars, counts):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 1,
             str(count), ha="center", va="bottom", fontsize=9)
plt.tight_layout()
plt.show()
```



Key Observations:

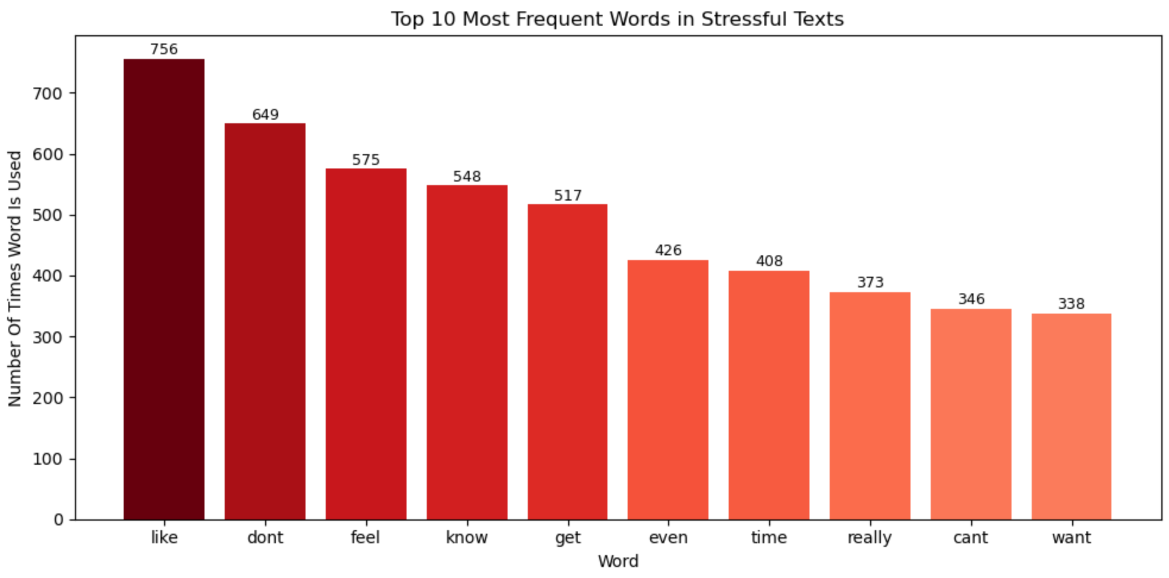
The most frequent word is “like” (425), followed by “time” (349) and “get” (335). Other highly used words include “know” (308), “dont” (285), “really” (277), “one” (276), “people” (248), “years” (240), and “help” (230). The distribution shows that non-stressful texts rely heavily on general conversational words rather than emotionally charged or crisis-specific terms. Most of these words are common in everyday dialogue and reflective writing.

Possible Insight:

The vocabulary pattern indicates that non-stressful texts are conversational, reflective and socially oriented. Words like time, years and back point to temporal framing, where users recount experiences or reflect on the past. Meanwhile, people and help highlight community engagement and supportive interactions, showing that non-stressful texts often involve advice-seeking, offering help or discussing relationships in a positive or neutral tone.

In [41]:
```
# Plotting Top 10 most frequent words for Stressful Texts

# Normalize counts for color intensity (shades of red)
counts = df_stress["Count"].values
colors = plt.cm.Reds(counts / counts.max())
```

```python
# Plot bar chart
plt.figure(figsize=(10, 5))
bars = plt.bar(df_stress["Word"], counts, color=colors)
plt.title("Top 10 Most Frequent Words in Stressful Texts")
plt.xlabel("Word")
plt.ylabel("Number Of Times Word Is Used")
for bar, count in zip(bars, counts):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 1,
             str(count), ha="center", va="bottom", fontsize=9)
plt.tight_layout()
plt.show()
```



Key Observations:

The most frequent word is “like” (756), followed by “dont” (649) and “feel” (575). Other highly used words include “know” (548), “get” (517), “even” (426), “time” (408), “really” (373), “cant” (346), and “want” (338). Compared to the non-stressful texts chart, stressful texts show a much higher frequency overall - for example, like appears almost twice as often in stressful texts (756 vs 425). The vocabulary also shifts toward emotionally charged and negative expressions. These words directly convey distress, uncertainty or limitation.

Possible Insight:

Stressful texts are characterized by higher intensity and repetition of emotionally loaded words. The prominence of dont and cant highlights themes of restriction, helplessness or barriers, while feel and want emphasize emotional states and unmet needs. The frequent use of even and still suggests ongoing struggles or frustration, reinforcing the sense of persistence in distress.

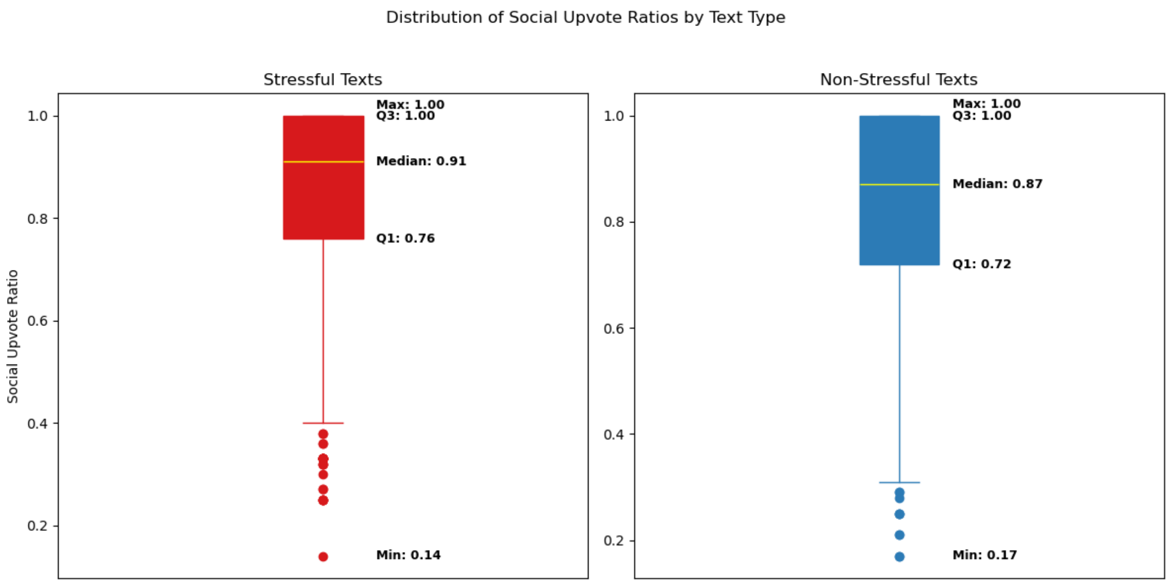
Visualization 4: Exploring Social Upvote Ratio Distributions On Stressful And Non-Stressful Texts

The social upvote ratio is a metric that reflects the relative approval of a post or text by the online community. It is calculated as the proportion of upvotes to the total number of votes (upvotes + downvotes) that a post receives. A higher social upvote ratio indicates that a larger proportion of users found the post helpful, relatable or valuable, whereas a lower ratio may suggest disagreement, irrelevance or negative reception.

Stressful texts, which often describe personal struggles, distress or sensitive experiences, may elicit more empathetic responses from readers, potentially resulting in higher upvote ratios if the content resonates with the community. Conversely, non-stressful texts, which may be neutral, informational or casual, might not engage the same emotional response and their social upvote ratios could vary differently.

A boxplot is employed, with one boxplot for stressful texts and another for non-stressful texts. Boxplots provide a concise summary of the distribution, highlighting key statistics such as the median, interquartile range, minimum and maximum values. By comparing the boxplots, we can visually assess differences in central tendency, spread and potential outliers between stressful and non-stressful texts. The primary aim is to determine whether stressful posts consistently receive higher community engagement as measured by the social upvote ratio.

In [43]:
```
# Separate social upvote ratios by label
stress_upvotes = dreaddit_train[dreaddit_train["Label"] == 1]["Upvote Proportior
non_stress_upvotes = dreaddit_train[dreaddit_train["Label"] == 0]["Upvote Propor

def annotate_boxplot(bp, data, ax):
    stats = np.percentile(data, [0, 25, 50, 75, 100])
    labels = ["Min", "Q1", "Median", "Q3", "Max"]
    prev_y = None
    for i, (y, label) in enumerate(zip(stats, labels)):
        if prev_y is not None and y == prev_y:
            if label == "Max":
                y_text = y + 0.02
            elif label == "Q3":
                y_text = y - 0.02
            else:
                y_text = y
        else:
            y_text = y
        ax.text(1.1, y_text, f"{label}: {y:.2f}", va='center', fontsize=9, fontw
        prev_y = y

# Plot side-by-side boxplots
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
colors = ["#d7191c", "#2c7bb6"]

# Stressful Texts
bp1 = axes[0].boxplot(stress_upvotes, patch_artist=True, boxprops=dict(facecolor
                      whiskerprops=dict(color=colors[0]), capprops=dict(color=cc

axes[0].set_title("Stressful Texts")
axes[0].set_ylabel("Social Upvote Ratio")
axes[0].set_xticks([])
annotate_boxplot(bp1, stress_upvotes, axes[0])

# Non-Stressful Texts
bp2 = axes[1].boxplot(non_stress_upvotes, patch_artist=True, boxprops=dict(facec
                      whiskerprops=dict(color=colors[1]), capprops=dict(color=cc
axes[1].set_title("Non-Stressful Texts")
axes[1].set_xticks([])
annotate_boxplot(bp2, non_stress_upvotes, axes[1])
plt.suptitle("Distribution of Social Upvote Ratios by Text Type")
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



Key Observations:

1. Both stressful and non-stressful texts reach the same maximum upvote ratio of 1.0, meaning that some posts in each category were universally approved by the community. However, the median upvote ratio is slightly higher for stressful texts (0.91) compared to non-stressful texts (0.87). Similarly, the first quartile (Q1) is higher for stressful texts (0.76) than for non-stressful texts (0.72), suggesting that even the lower-performing stressful posts tend to receive stronger approval than their non-stressful counterparts.

2. The minimum values show a small difference: stressful texts dip as low as 0.14 while non-stressful texts bottom out at 0.17. This indicates that while both categories can receive poor reception, stressful texts have a slightly wider spread in community response. Overall, the distribution suggests that stressful texts are more consistently well-received with higher central tendency values.

Possible Insights:

The data supports the idea that stressful texts elicit more empathetic and supportive responses from the community. Posts describing personal struggles or distress may resonate more deeply, prompting users to upvote as a gesture of solidarity or encouragement. This explains why stressful texts show higher median and quartile values in their upvote ratios. Non-stressful texts may be more neutral, informational or casual. While they can still achieve high approval, they do not consistently generate the same empathetic engagement. Their slightly lower median and quartile values suggest that readers may be less compelled to upvote unless the content is particularly useful or relatable.

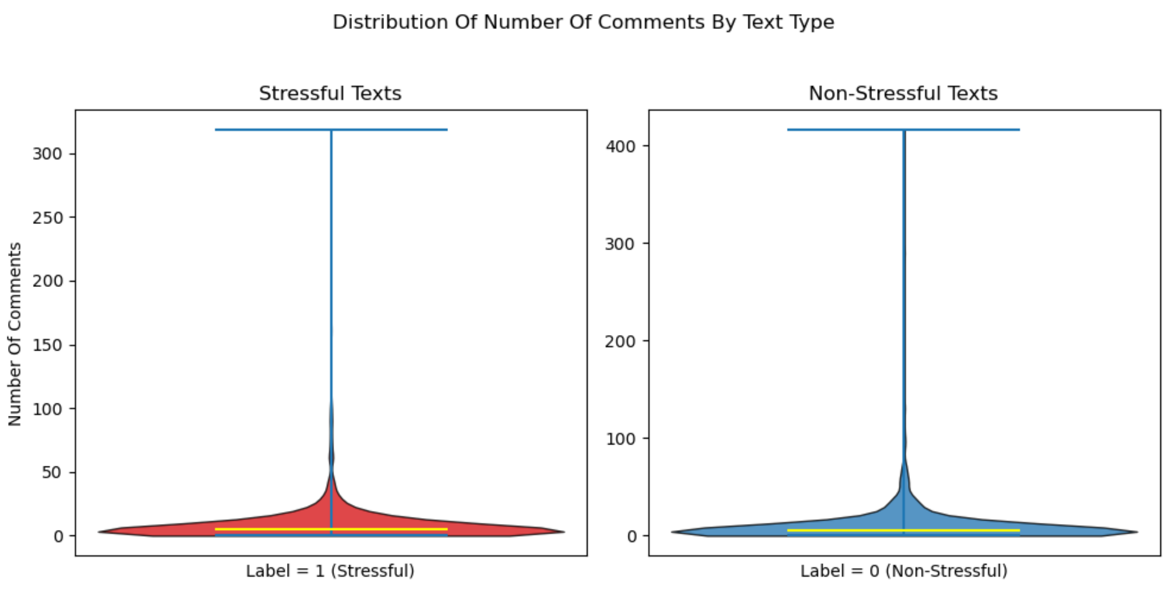
Visualization 5: Exploring Number Of Comments On Stressful And Non-Stressful Texts

The number of comments associated with a post provides an additional social engagement signal that may help distinguish between stressful and non-stressful texts. Posts that express distress, anxiety or emotional vulnerability are more likely to elicit responses from the community such as offering advice, emotional support or shared experiences. In contrast, non-stressful texts may attract fewer comments or more neutral interactions.

To examine this relationship, we visualize the distribution of the number of comments for stressful and non-stressful texts using a violin plot. Compared to a boxplot, a violin plot captures not only summary statistics (such as the median and interquartile range) but also the full distributional shape, making it particularly suitable for identifying skewness, multimodality and extreme values. This visualization allows us to assess whether stressful texts tend to generate a higher volume of comments or exhibit greater variability in engagement, thereby offering deeper insight into how users respond to different types of content.

Visualization 5 Part 1: Distribution Of Number Of Comments By Text Type (Absolute Number)

In [45]:
```
# Extract numeric data
stress_num_comments = dreaddit_train[dreaddit_train["Label"] == 1]["Number Of Co
non_stress_num_comments = dreaddit_train[dreaddit_train["Label"] == 0]["Number O
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# Stressful texts (Label = 1)
vp1 = axes[0].violinplot(stress_num_comments, showmedians=True)
for body in vp1['bodies']:
    body.set_facecolor("#d7191c")
    body.set_edgecolor("black")
    body.set_alpha(0.8)
vp1['cmedians'].set_color("yellow")
axes[0].set_title("Stressful Texts")
axes[0].set_ylabel("Number Of Comments")
axes[0].set_xticks([])
axes[0].set_xlabel("Label = 1 (Stressful)")

# Non-stressful texts (Label = 0)
vp2 = axes[1].violinplot(non_stress_num_comments, showmedians=True)
for body in vp2['bodies']:
    body.set_facecolor("#2c7bb6")
    body.set_edgecolor("black")
    body.set_alpha(0.8)
vp2['cmedians'].set_color("yellow")
axes[1].set_title("Non-Stressful Texts")
axes[1].set_xticks([])
axes[1].set_xlabel("Label = 0 (Non-Stressful)")

plt.suptitle("Distribution Of Number Of Comments By Text Type")
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



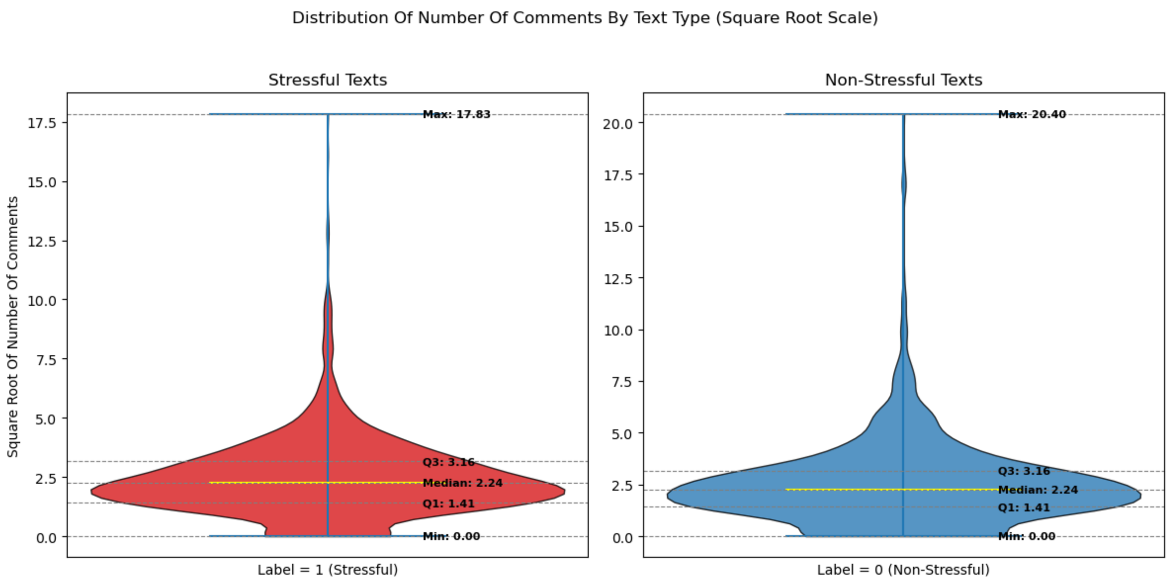
The violin plot generated highlights a key limitation: **the raw distribution of comment counts is extremely skewed**. There are several texts in both stressful and non-stressful texts that have many comments which make the maximum have much higher values. This makes the visualization unsuitable for deeper analysis because it hides subtle differences in spread and central tendency.

A better next step is to experiment with alternative transformations such as the **square root** transform. Unlike the log, the square root reduces skew while still preserving more variation in the mid-range values. This helps us to see differences in quartiles and medians without letting extreme outliers dominate the scale. By doing so, the interpretability of the raw counts will be retained while making the distribution shape clearer and the summary statistics more meaningful for comparison.

Visualization 5 Part 2: Distribution Of Number Of Comments By Text Type (Square Root Transforme)

In [47]:
```python
# Apply square root transform to reduce skew
stress_num_comments_sqrt = np.sqrt(stress_num_comments)
non_stress_num_comments_sqrt = np.sqrt(non_stress_num_comments)
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

def add_summary_stats(ax, data, xpos=1):
    stats = {"Median": np.median(data), "Q1": np.percentile(data, 25), "Q3": np.
    for key, val in stats.items():
        ax.axhline(val, color="gray", linestyle="--", linewidth=0.8)
        ax.text(xpos + 0.1, val, f"{key}: {val:.2f}", va="center", fontsize=8, f

# Stressful texts (Label = 1)
vp1 = axes[0].violinplot(stress_num_comments_sqrt, showmedians=True)
for body in vp1['bodies']:
    body.set_facecolor("#d7191c")
    body.set_edgecolor("black")
    body.set_alpha(0.8)
vp1['cmedians'].set_color("yellow")
axes[0].set_title("Stressful Texts")
axes[0].set_ylabel("Square Root Of Number Of Comments")
axes[0].set_xticks([])
axes[0].set_xlabel("Label = 1 (Stressful)")
add_summary_stats(axes[0], stress_num_comments_sqrt)
```

```
# Non-stressful texts (Label = 0)
vp2 = axes[1].violinplot(non_stress_num_comments_sqrt, showmedians=True)
for body in vp2['bodies']:
    body.set_facecolor("#2c7bb6")
    body.set_edgecolor("black")
    body.set_alpha(0.8)
vp2['cmedians'].set_color("yellow")
axes[1].set_title("Non-Stressful Texts")
axes[1].set_xticks([])
axes[1].set_xlabel("Label = 0 (Non-Stressful)")
add_summary_stats(axes[1], non_stress_num_comments_sqrt)

plt.suptitle("Distribution Of Number Of Comments By Text Type (Square Root Scale
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



Key Observations:

The square-root scaled violin plots show that both stressful and non-stressful texts share nearly identical quartiles: **Q1 ≈ 1.41, median ≈ 2.24, and Q3 ≈ 3.16**, with a minimum of 0. This indicates that the bulk of the data is concentrated at low comment counts, regardless of text type. The maximum values differ slightly, with **stressful texts reaching ≈ 17.8 and non-stressful texts ≈ 20.4**. The violins themselves are narrow at the base, widen around the median and taper toward the maximum, reflecting a skewed distribution with many low-comment texts and a few extreme outliers.

Possible Insights:

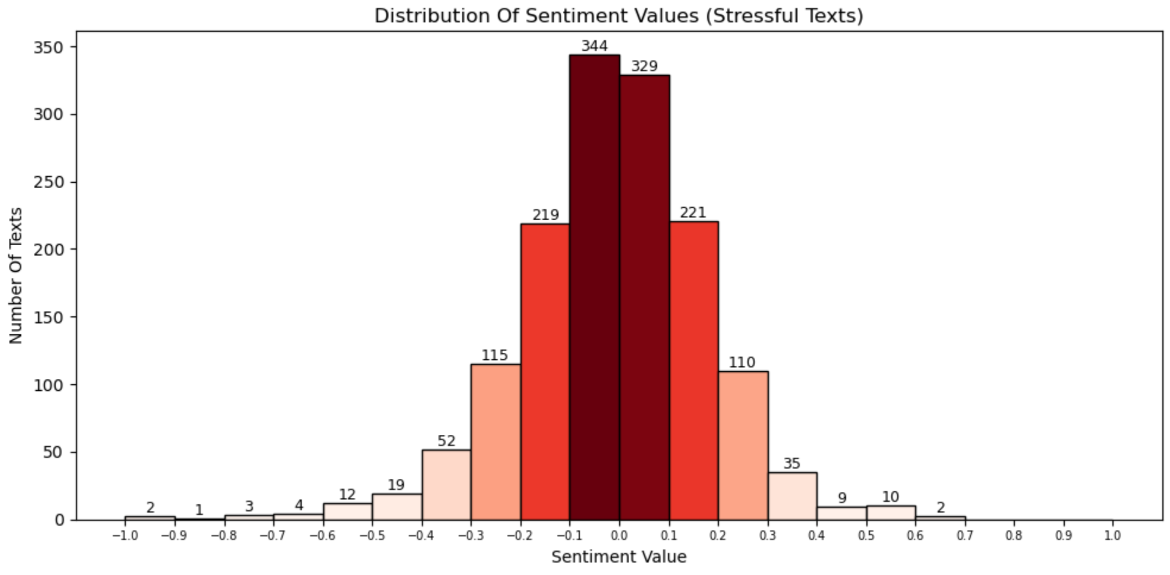
The identical quartiles likely result from two factors: the square-root transformation, which reduces skew but can map multiple discrete values to the same transformed positions and the inherently discrete nature of comment counts. The higher maximum for non-stressful texts contradicts the hypothesis that stressful texts attract more engagement. This discrepancy may be due to outliers: a few non-stressful posts could have gone viral or received unusually high attention because of topic popularity, posting time or community dynamics. Since maximum values are highly sensitive to single extreme cases, they are not reliable indicators of overall engagement.

Visualization 6: Plotting The Distribution Of Sentiment Values From Stressful And Non Stressful Text

This visualization aims to examine the distribution of sentiment values in stressful and non-stressful texts, with particular emphasis on identifying the modal sentiment ranges for each group. Sentiment scores provide a numerical summary of emotional tone, ranging from negative to positive values. By observing where sentiment values concentrate most frequently, we can assess whether stressful texts tend to cluster around more negative sentiment ranges while non-stressful texts may center around neutral or slightly positive values. Understanding these differences is important because sentiment polarity often reflects underlying emotional distress, frustration or anxiety which are key indicators of stress.

A **histogram** is used for each text category with a fixed bin width of 0.1, spanning the full sentiment range from -1.0 to +1.0. Using a constant bin interval ensures fair and consistent comparison between stressful and non-stressful texts, allowing differences in modal ranges and distribution shapes to be clearly observed. Histograms are particularly suitable here because they reveal both the frequency and spread of sentiment values, making it easy to detect skewness, clustering and overlaps between the two groups.

```
In [49]: # Filter sentiment values by label
stressful_sentiment = dreaddit_train.loc[dreaddit_train["Label"] == 1, "Sentimer
non_stressful_sentiment = dreaddit_train.loc[dreaddit_train["Label"] == 0, "Sent

bins = np.arange(-1.0, 1.1, 0.1)
plt.figure(figsize=(10, 5))
counts, bin_edges, patches = plt.hist(stressful_sentiment, bins=bins, density=Fa

# Apply red color scale with intensity based on counts
max_count = max(counts)
for count, patch in zip(counts, patches):
    patch.set_facecolor(plt.cm.Reds(count / max_count))
    patch.set_edgecolor("black")
for count, patch in zip(counts, patches):
    if count > 0:
        plt.text(patch.get_x() + patch.get_width() / 2, count, int(count),
                 ha="center", va="bottom", fontsize=9)
plt.title("Distribution Of Sentiment Values (Stressful Texts)")
plt.xlabel("Sentiment Value")
plt.ylabel("Number Of Texts")
plt.xticks(bins, fontsize = 7)
plt.tight_layout()
plt.show()
```



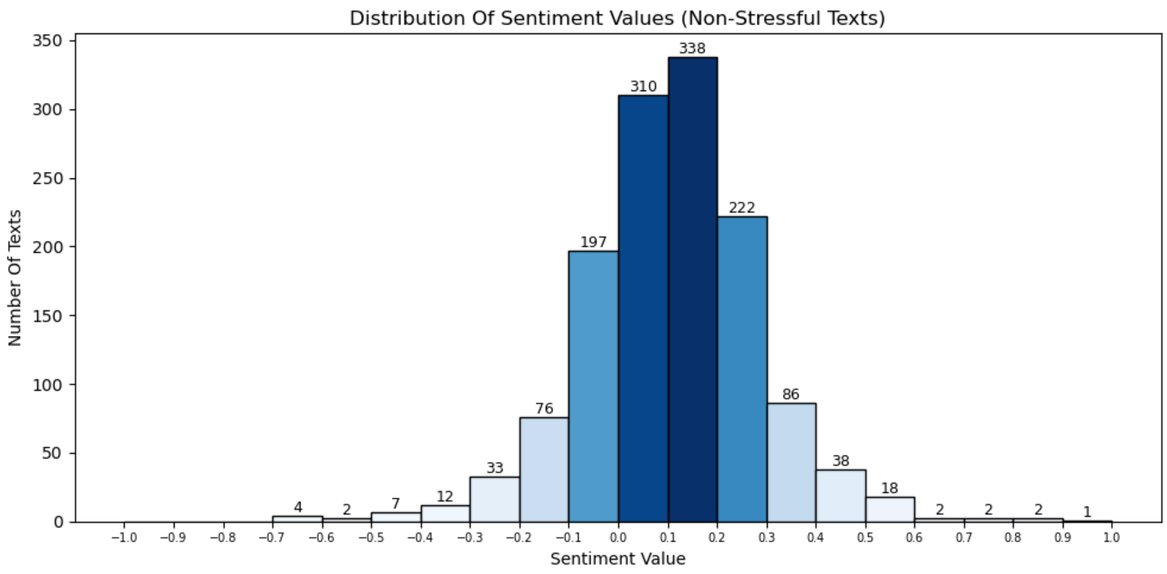
Key Observations:

The histogram of sentiment values for **stressful texts** shows a strong clustering around neutral to slightly negative sentiment. The highest frequencies occur at sentiment values of **between -0.1 and 0.0** (344 texts), followed closely by **between 0.0 and 0.1** (329 texts). This indicates that most stressful texts are not extremely polarized but instead hover near the neutral range. At the extremes, very few texts fall below -0.5 or above 0.5, with counts dropping sharply as sentiment moves toward -1.0 or +1.0.

Possible Insights:

The concentration of stressful texts around neutral sentiment suggests that stress does not necessarily manifest in overtly negative language. Instead, many stressful posts may be expressed in a balanced or matter of fact tone, which sentiment analysis interprets as neutral. The scarcity of extreme values implies that highly emotional or strongly positive/negative language is rare in this dataset. This could be due to the nature of the platform or community, where users describe stressful experiences in a restrained way rather than venting with extreme sentiment. The slight skew toward negative values aligns with expectations that stressful texts lean towards a more negative sentiment.

In [51]:
```
bins = np.arange(-1.0, 1.1, 0.1)
plt.figure(figsize=(10, 5))
counts, bin_edges, patches = plt.hist(non_stressful_sentiment, bins=bins, densit

# Apply blue color scale with intensity based on counts
max_count = max(counts)
for count, patch in zip(counts, patches):
    patch.set_facecolor(plt.cm.Blues(count / max_count))
    patch.set_edgecolor("black")
for count, patch in zip(counts, patches):
    if count > 0:
        plt.text(patch.get_x() + patch.get_width() / 2, count, int(count),
                 ha="center", va="bottom", fontsize=9)

plt.title("Distribution Of Sentiment Values (Non-Stressful Texts)")
plt.xlabel("Sentiment Value")
plt.ylabel("Number Of Texts")
plt.xticks(bins, fontsize = 7)
plt.tight_layout()
plt.show()
```



Key Observations:

The histogram of sentiment values for non-stressful texts shows a concentration around neutral to slightly positive sentiment. The highest frequencies occur **between 0.1 and 0.2** (338 texts), followed closely by **between 0.0 and 0.1** (310 texts), indicating that most non-stressful texts lean toward neutrality or mild positivity. Negative sentiment values are present but far less frequent, with counts dropping sharply below -0.2. The distribution is roughly bell-shaped, tapering off toward both extremes, but with a noticeable skew toward the positive side compared to stressful texts.

Possible Insights:

The clustering around neutral and slightly positive values suggests that non-stressful texts are often expressed in balanced or optimistic tones, which sentiment analysis captures as positive. This contrasts with stressful texts, which clustered around neutral to slightly negative values. The difference implies that stress influences language tone, but not in an extreme way. Stressful texts avoid strong negativity, while non-stressful texts lean gently positive. The higher peak at 0.1 for non-stressful texts may reflect community norms of supportive or casual communication, where neutral posts sometimes carry a slight positive framing.

Visualization 7: Exploring LIWC Linguistic Features and Their Relationships with Stressful vs. Non-Stressful Texts

The Linguistic Inquiry and Word Count (LIWC) tool is widely used in computational linguistics and psychological research to quantify the presence of specific psychological, emotional and cognitive processes in text. LIWC categorizes words into several meaningful psychological dimensions, such as affective states (anxiety, sadness, anger), cognitive processes (causation, insight, certainty), self-focus (use of first-person pronouns), and social or situational stressors (family, work-related terms).

In the context of analyzing stressful versus non-stressful texts, these LIWC factors may reveal meaningful patterns. For example, texts with higher anxiety or anger scores may be more likely to be labeled as stressful whereas texts with higher positive affect or cognitive processing words might correlate with non-stressful texts.

In this visualization, we focus on six main categories:

1. **LIWC Affective Features** - capturing emotions such as anxiety, sadness, anger and positive affect.
2. **LIWC Cognitive Processing Features** - including analytical thinking, causation, certainty and insight.
3. **LIWC Self-Focus** - measuring the degree to which texts use first-person singular or plural pronouns, reflecting self-orientation or self-involvement.
4. **LIWC Social and Situational Stressors** - representing references to family, friends, work or other social stress contexts.
5. **LIWC Psychological Style and Social Expression Features** - representing references to social confidence, authenticity and tone
6. **DAL Semantic-Emotional Content Features** - includes features such as emotional activation, mental imagery and emotional valence

For each of these categories, we will:

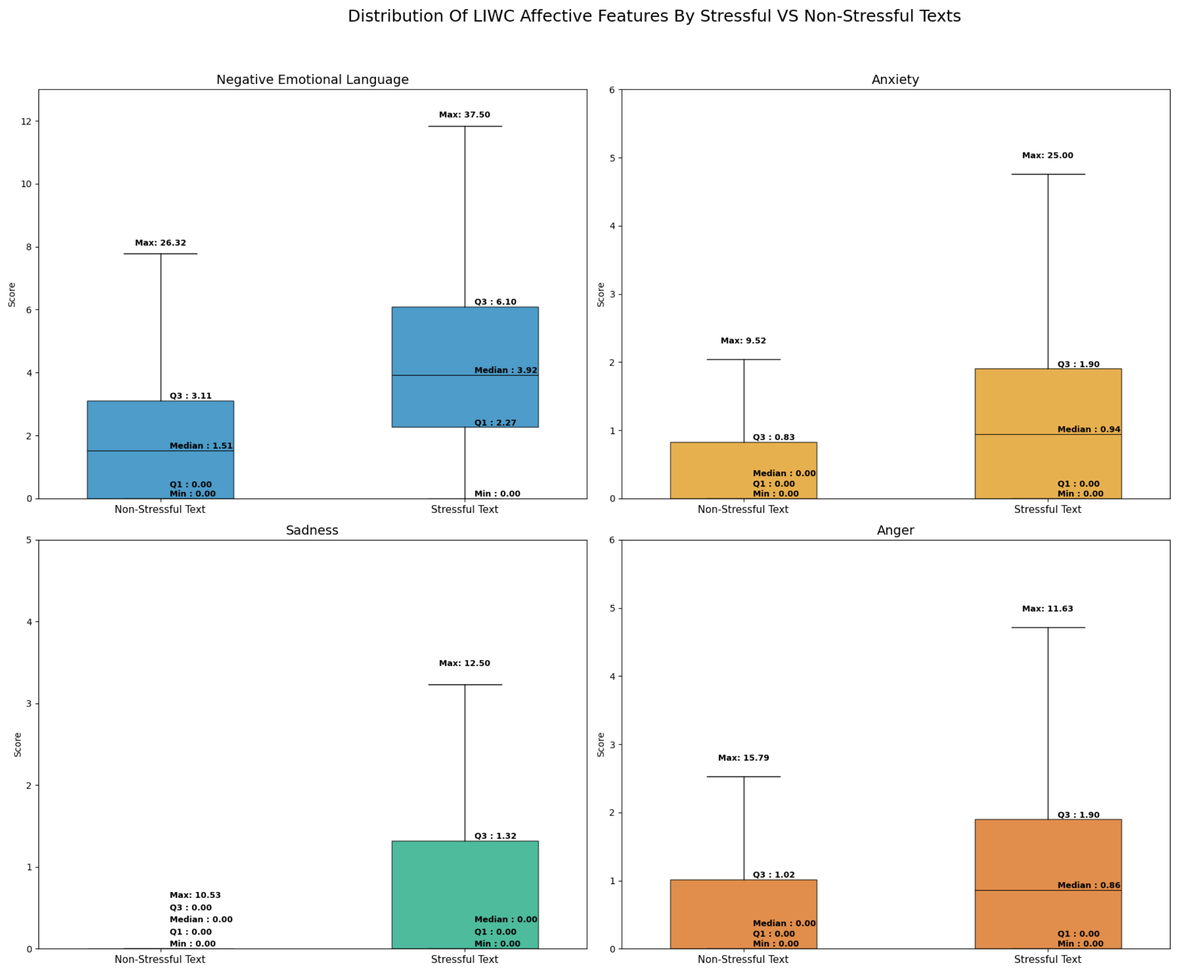
1. Use boxplots to visualize the distribution of LIWC scores across stressful and non-stressful texts, providing insight into how different psychological or linguistic factors vary between the two groups.
2. Plot correlations between each subcategory within the LIWC group and the label, allowing us to quantify and compare the strength of association between linguistic features and text stressfulness.

This combined approach provides both a descriptive and quantitative understanding of how psychological and linguistic markers manifest in stressful versus non-stressful communication, helping to highlight the most informative features for subsequent modeling or analysis.

Visualization 7 Part 1: LIWC Affective Features

First, we will create boxplots to visualize the distribution of LIWC scores across stressful and non-stressful texts for the LIWC affective features, which include: **Negative Emotional Language**, **Anxity**, **Sadness** and **Anger**.

In [53]:
```python
from matplotlib.patches import Patch

# LIWC Affective Features with cleaned column names
liwc_features = ["Negative Emotional Language", "Anxiety", "Sadness", "Anger"]
colors = sns.color_palette("colorblind", len(liwc_features))
fig, axes = plt.subplots(2, 2, figsize=(20, 15))
axes = axes.flatten()

for i, feature in enumerate(liwc_features):
    ax = axes[i]
    data = [dreaddit_train[dreaddit_train['Label'] == 0][feature].dropna(), drea
    positions = [1, 2.25]
    bp = ax.boxplot(data, positions=positions, widths=0.6, patch_artist=True, sh
    for patch in bp['boxes']:
        patch.set_facecolor(colors[i])
        patch.set_edgecolor("black")
        patch.set_alpha(0.7)
    whisker_tops = {}
    true_max_vals = {}
    for j, vals in enumerate(data):
        stats = {"Min": np.min(vals), "Q1": np.percentile(vals, 25), "Median": r
        true_max_vals[j] = np.max(vals)
        whisker_val = np.max([w.get_ydata().max() for w in bp['whiskers'] if w.g
        whisker_tops[j] = whisker_val
        seen_vals = {}
        for k, (stat_name, val) in enumerate(stats.items()):
            offset = 0
            if val in seen_vals:
                offset = 0.15 * len(seen_vals[val])
                seen_vals[val].append(stat_name)
            else:
                seen_vals[val] = [stat_name]
            if feature == "Negative Emotional Language":
                if stat_name == "Q1" and stats["Q1"] == stats["Min"]:
                    offset += 0.15
            if j == 0:
                x_pos = positions[j] + 0.04
            else:
                x_pos = positions[j] + 0.04
            ax.text(x_pos, val + offset, f"{stat_name} : {val:.2f}", ha='left',
                    va='bottom', fontsize=9, color="black", fontweight="bold")
        if feature == "Sadness" and j == 0:
            ax.text(positions[j] + 0.04, stats["Q3"] + 0.6, f"Max: {true_max_val
                    ha='left', va='bottom', fontsize=9, color="black", fontweigh
        else:
            ax.text(positions[j], whisker_tops[j] + 0.2, f"Max: {true_max_vals[j
                    ha='center', va='bottom', fontsize=9, color="black", fontwei
    ax.set_xticks([1, 2.25])
    ax.set_xticklabels(["Non-Stressful Text", "Stressful Text"], fontsize=11)
    ax.set_title(feature, fontsize=14)
    ax.set_ylabel("Score")
    ax.set_ylim(0, np.ceil(max(whisker_tops.values()) + 1))

fig.suptitle("Distribution Of LIWC Affective Features By Stressful VS Non-Stress
plt.subplots_adjust(hspace=0.5)
plt.tight_layout(rect=[0, 0, 0.9, 0.95])
plt.show()
```



Key Observations:

Negative Emotional Language stands out with the highest values overall, especially in stressful texts where the median and upper quartile are clearly elevated compared to other categories. **Anxiety, Sadness, and Anger** show much lower distributions, with many values clustering near zero for non-stressful texts. Sadness and Anger in particular reveal very compressed ranges in non-stressful contexts, suggesting that these emotions are far less frequently expressed compared to negative emotional language more broadly. **Anxiety**, while generally low, shows a wider spread in stressful texts, indicating that stress tends to amplify variability in anxious language use.

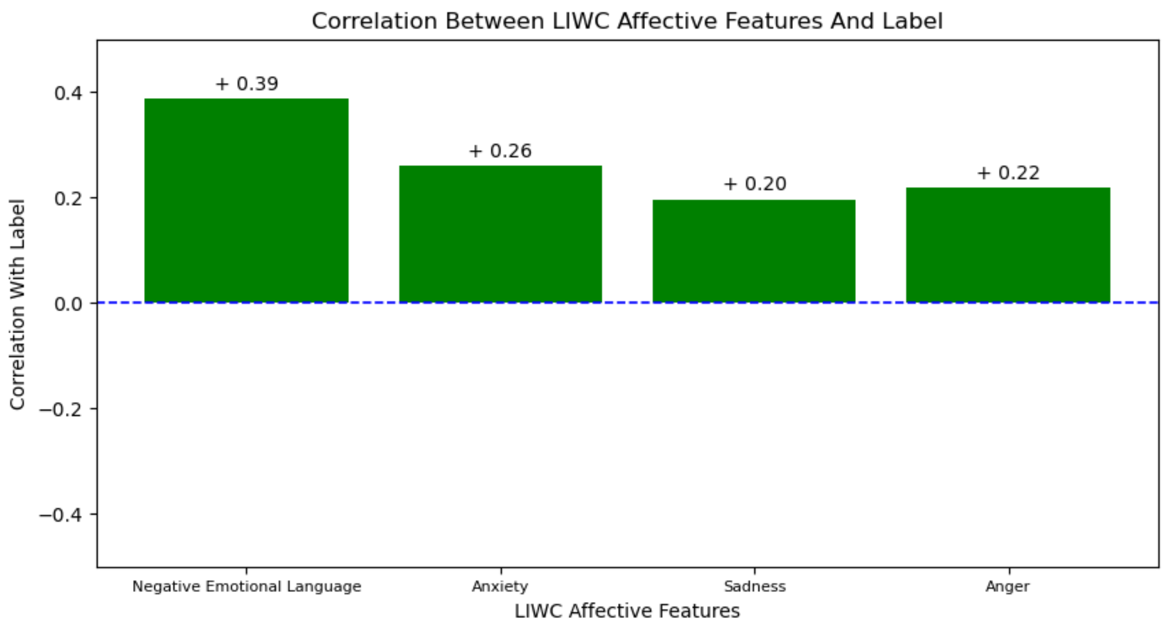
Differences between stressful and non-stressful texts:

Across all four categories, stressful texts consistently exhibit higher medians, quartiles and maximum values than non-stressful texts. For example, stressful texts in Negative Emotional Language reach a median of nearly 4 compared to about 1.5 in non-stressful texts, while the upper quartile doubles from around 3 to over 6. This pattern suggests that stress not only increases the intensity of emotional language but also broadens its variability, making emotional expression more pronounced and diverse compared to non-stressful contexts.

Now, we will explore the correlations between each subcategory within the LIWC group of affective features and the label, allowing us to quantify and compare the strength of association between linguistic features and text stressfulness.

In [55]:

```python
features = ["Negative Emotional Language", "Anxiety", "Sadness", "Anger"]
correlations = dreaddit_train[features + ['Label']].corr()['Label'].drop('Label'
colors = ['green' if val >= 0 else 'red' for val in correlations]
plt.figure(figsize=(10, 5))
bars = plt.bar(correlations.index, correlations.values, color=colors)

for bar, val in zip(bars, correlations.values):
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01 if val >=
             ha='center', va='bottom' if val >= 0 else 'top', fontsize=10)

plt.axhline(0, color='blue', linewidth=1.2, linestyle='--')
plt.xticks(fontsize=8)
plt.ylabel("Correlation With Label")
plt.xlabel("LIWC Affective Features")
plt.title("Correlation Between LIWC Affective Features And Label")
plt.ylim(-0.5, 0.5)
plt.show()
```



Key Observations:

Negative Emotional Language shows the strongest correlation with the stress label, around +0.39, which suggests that texts containing more negative emotional words are most predictive of being stressful. **Anxiety** follows with a moderate positive correlation (+0.26), while **Anger** (+0.22) and **Sadness** (+0.20) are slightly weaker but still positive. This ordering indicates that while all affective features contribute to distinguishing stressful texts, broad negative emotional language is the most dominant signal.

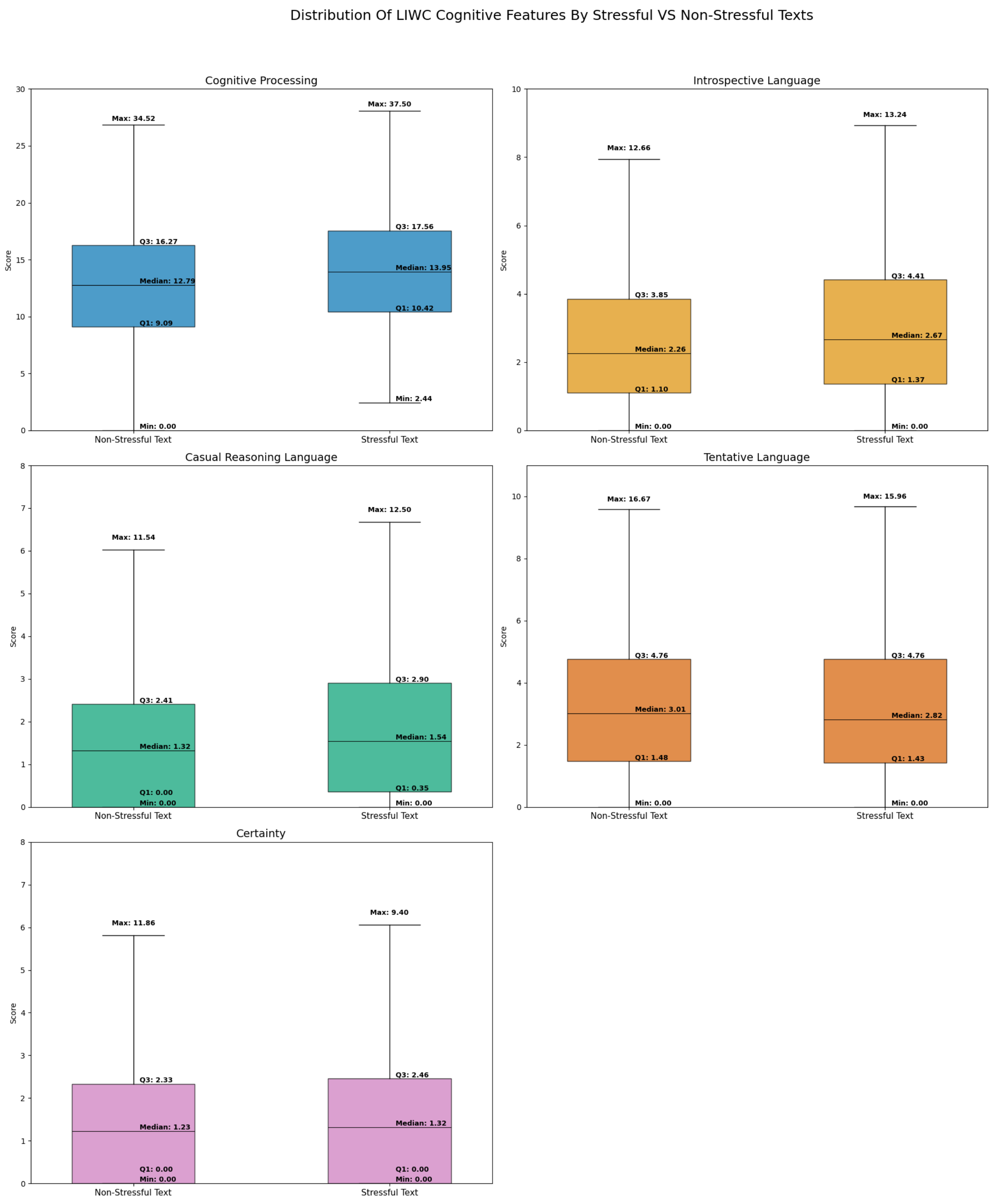
Possible Insights:

The fact that all correlations are positive means that higher values of these affective features consistently align with stressful texts, while lower values align with non-stressful texts. The relatively stronger correlation for Negative Emotional Language suggests that stressful texts are not only more likely to contain specific emotions like anxiety or sadness, but also a general increase in negative affective vocabulary overall. Meanwhile, the weaker but still positive correlations for Sadness and Anger imply that these emotions are present but less consistently predictive compared to anxiety or broad negativity.

Visualization 7 Part 2: LIWC Cognitive Processing Features

Next, we will create boxplots to visualize the distribution of LIWC scores across stressful and non-stressful texts for the LIWC Cognitive Processing features, which include: **Cognitive Processing**, **Introspective Language**, **Casual Reasoning Language**, **Tentative Language** and **Certainty**.

In [57]:
```
# LIWC Cognitive / Language Features
cog_features = ['Cognitive Processing', 'Introspective Language',  'Casual Reaso
colors = sns.color_palette("colorblind", len(cog_features))
fig, axes = plt.subplots(3, 2, figsize=(20, 22))
axes = axes.flatten()

for i, feature in enumerate(cog_features):
    ax = axes[i]
    data = [dreaddit_train[dreaddit_train['Label'] == 0][feature].dropna(), drea
    positions = [1, 2.25]
    bp = ax.boxplot(data, positions=positions, widths=0.6, patch_artist=True, sh
    for patch in bp['boxes']:
        patch.set_facecolor(colors[i])
        patch.set_edgecolor("black")
        patch.set_alpha(0.7)
    whisker_tops = {}
    true_max_vals = {}
    for j, vals in enumerate(data):
        stats = {"Min": np.min(vals), "Q1": np.percentile(vals, 25), "Median": r
        true_max_vals[j] = np.max(vals)
        whisker_val = np.max([w.get_ydata().max() for w in bp['whiskers'] if w.g
        whisker_tops[j] = whisker_val
        for stat_name, val in stats.items():
            offset = 0
            if stat_name == "Q1" and stats["Min"] == stats["Q1"]:
                offset += 0.25
            x_pos = positions[j] + 0.03
            ax.text(x_pos, val + offset, f"{stat_name}: {val:.2f}", ha='left', v
        ax.text(positions[j], whisker_tops[j] + 0.2, f"Max: {true_max_vals[j]:.2
                ha='center', va='bottom', fontsize=9, color="black", fontweight=
    ax.set_xticks([1, 2.25])
    ax.set_xticklabels(["Non-Stressful Text", "Stressful Text"], fontsize=11)
    ax.set_title(feature, fontsize=14)
    ax.set_ylabel("Score")
    ax.set_ylim(0, np.ceil(max(whisker_tops.values()) + 1))

axes[-1].axis('off')
fig.suptitle("Distribution Of LIWC Cognitive Features By Stressful VS Non-Stress
plt.subplots_adjust(hspace=0.5)
plt.tight_layout(rect=[0, 0, 0.9, 0.95])
plt.show()
```



Key Observations:

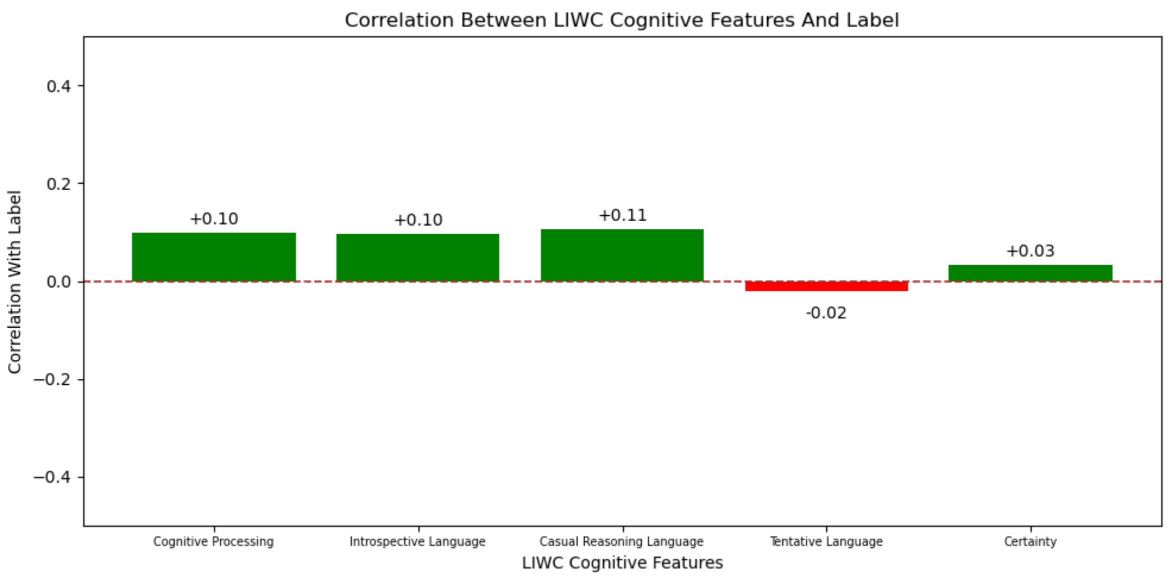
Cognitive Processing shows the highest overall values among the features, with medians around 13-14 and upper quartiles reaching 16-18. This suggests that both stressful and non-stressful texts are rich in cognitive markers, reflecting reasoning and analytical language. **Introspective Language** and **Tentative Language** occupy a middle range, with medians around 2-3 and upper quartiles near 4-5, indicating moderate use of self-reflective and uncertain phrasing. **Casual Reasoning** and **Certainty** are lower overall, with medians closer to 1 and upper quartiles around 2-3, suggesting that explicit causal explanations and strong assertions are less frequent compared to broader cognitive processing or tentative expressions.

Differences between stressful and non-stressful texts:

Across most categories, stressful texts show slightly higher values than non-stressful texts. For example, Cognitive Processing in stressful texts has a higher median (13.99 vs. 12.77) and upper quartile (17.56 vs. 16.27), indicating more complex reasoning under stress. Certainty is slightly higher in stressful texts at the median but has a lower maximum, suggesting that while stressful texts may include more frequent assertions, they avoid extreme certainty compared to non-stressful texts. Tentative Language remains nearly identical between the two categories, showing that uncertainty is expressed at similar levels regardless of stress.

Now, we will explore the correlations between each subcategory within the LIWC group of cognitive features and the label.

In [59]:
```
cog_features = ['Cognitive Processing', 'Introspective Language', 'Casual Reasor
correlations = dreaddit_train[cog_features + ['Label']].corr()['Label'].drop('La
colors = ['green' if val >= 0 else 'red' for val in correlations]
plt.figure(figsize=(10, 5))
bars = plt.bar(correlations.index, correlations.values, color=colors)

for bar, val in zip(bars, correlations.values):
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01 if val >=
             ha='center', va='bottom' if val >= 0 else 'top', fontsize=10)

plt.axhline(0, color='brown', linewidth=1.2, linestyle='--')
plt.xticks(fontsize=7)
plt.ylabel("Correlation With Label")
plt.xlabel("LIWC Cognitive Features")
plt.title("Correlation Between LIWC Cognitive Features And Label")
plt.ylim(-0.5, 0.5)
plt.tight_layout()
plt.show()
```



Key Observations:

Casual Reasoning Language shows the strongest positive correlation (+0.11), suggesting that when people are stressed, they tend to use more causal explanations in their language. **Cognitive Processing** and **Introspective Language** both follow closely at +0.10, indicating that stressed texts are slightly more likely to contain analytical or self-reflective language. **Certainty** has a weaker positive correlation (+0.03), implying that while stressed texts may include more assertive statements, the effect is minimal. **Tentative Language** stands out as the only feature with a negative correlation (-0.02), suggesting that uncertainty is slightly less common in stressful texts compared to non-stressful ones.

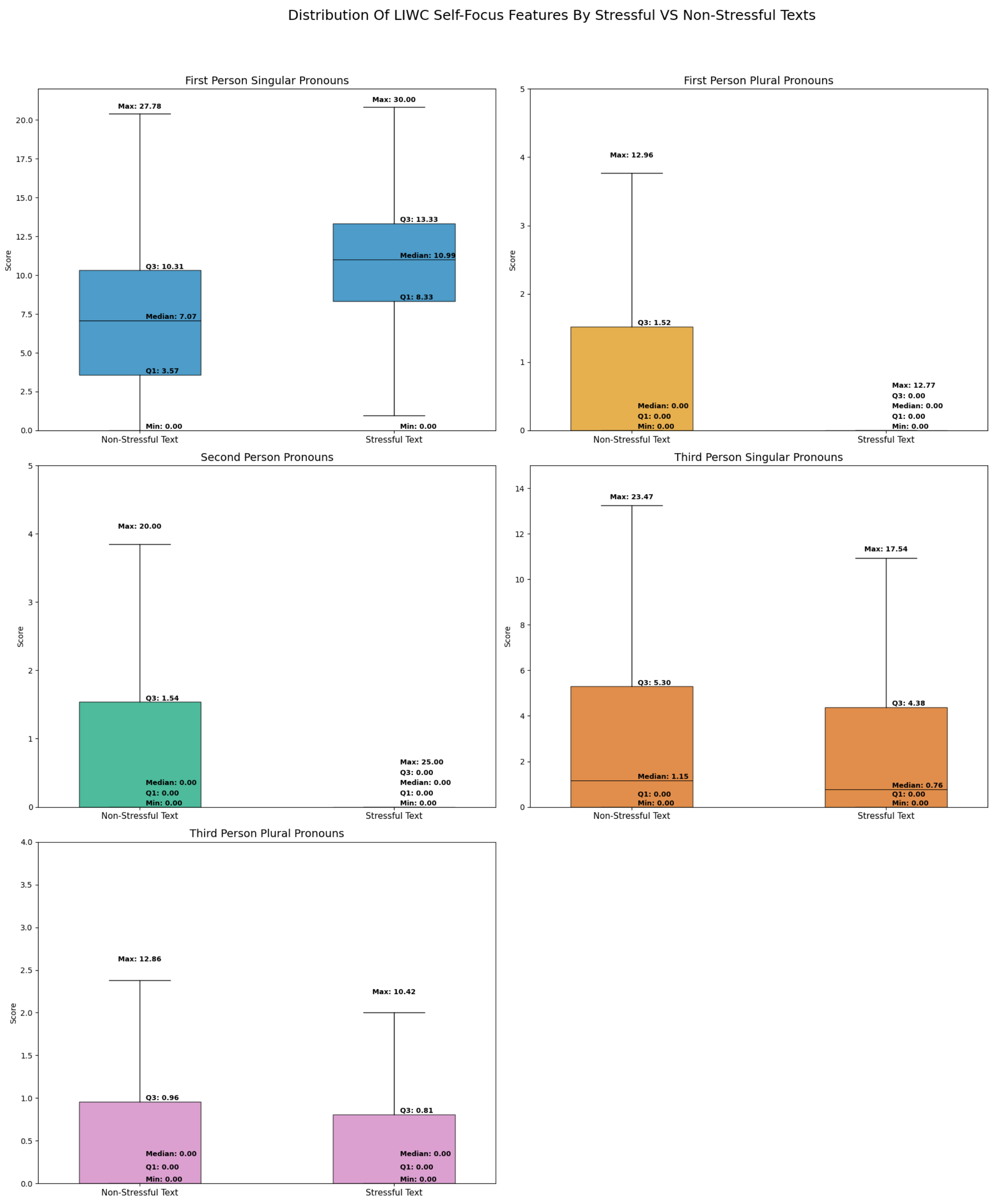
Possible Insights:

The overall pattern shows that stressful texts lean toward more reasoning, reflection and cognitive engagement, albeit with modest effect sizes. The positive correlations for Cognitive Processing, Introspective Language and Casual Reasoning suggest that stress prompts individuals to think more deeply and explain more often, possibly as a way of coping or rationalizing their experiences. The slight negative correlation for Tentative Language indicates that stressed individuals may be less likely to hedge or express uncertainty, instead favoring more direct or explanatory language.

Visualization 7 Part 3: LIWC Self Focus Features

Next, we will create boxplots to visualize the distribution of LIWC scores across stressful (label = 1) and non-stressful (label = 0) texts for the LIWC Self Focus features, which include: **First Person Singular Pronouns**, **First Person Plural Pronouns**, **Second Person Pronouns**, **Third Person Singular Pronouns** and **Third Person Plural Pronouns**.

In [61]:
```python
# LIWC Self-Focus / Pronoun Features
self_focus_features = ['First Person Singular Pronouns', 'First Person Plural Pr
                       'Third Person Plural Pronouns']
colors = sns.color_palette("colorblind", len(self_focus_features))
fig, axes = plt.subplots(3, 2, figsize=(20, 22))
axes = axes.flatten()

for i, feature in enumerate(self_focus_features):
    ax = axes[i]
    data = [dreaddit_train[dreaddit_train['Label'] == 0][feature].dropna(), drea
    positions = [1, 2.25]
    bp = ax.boxplot(data, positions=positions, widths=0.6, patch_artist=True, sh
    for patch in bp['boxes']:
        patch.set_facecolor(colors[i])
        patch.set_edgecolor("black")
        patch.set_alpha(0.7)
    whisker_tops = {}
    true_max_vals = {}
    for j, vals in enumerate(data):
        stats = {"Min": np.min(vals), "Q1": np.percentile(vals, 25), "Median": n
        true_max_vals[j] = np.max(vals)
        whisker_val = np.max([w.get_ydata().max() for w in bp['whiskers'] if w.g
        whisker_tops[j] = whisker_val
        stat_order = ["Min", "Q1", "Median", "Q3"]
        value_groups = {}
        for stat in stat_order:
          val = stats[stat]
          value_groups.setdefault(val, []).append(stat)
        for val, stat_list in value_groups.items():
          for idx, stat_name in enumerate(stat_list):
            offset = 0.15 * idx
            if feature == "Third Person Singular Pronouns":
                if stat_name == "Q1" and stats["Q1"] == stats["Min"]:
                    offset += 0.25
            x_pos = positions[j] + 0.03
            ax.text(x_pos, val + offset, f"{stat_name}: {val:.2f}", ha='left', v
        if stats["Q3"] == 0.00:
            stacked_count = len(value_groups.get(0.00, []))
            max_y = stats["Q3"] + (0.15 * stacked_count)
            max_x = positions[j] + 0.03
        else:
            max_y = whisker_tops[j] + 0.2
            max_x = positions[j]
        ax.text(max_x, max_y, f"Max: {true_max_vals[j]:.2f}", ha='left' if stats
                va='bottom', fontsize=9, color="black", fontweight="bold")
    ax.set_xticks([1, 2.25])
    ax.set_xticklabels(["Non-Stressful Text", "Stressful Text"], fontsize=11)
    ax.set_title(feature, fontsize=14)
    ax.set_ylabel("Score")
    ax.set_ylim(0, np.ceil(max(whisker_tops.values()) + 1))

axes[-1].axis('off')
fig.suptitle("Distribution Of LIWC Self-Focus Features By Stressful VS Non-Stres
plt.subplots_adjust(hspace=0.5)
plt.tight_layout(rect=[0, 0, 0.9, 0.95])
plt.show()
```



Key Observations:

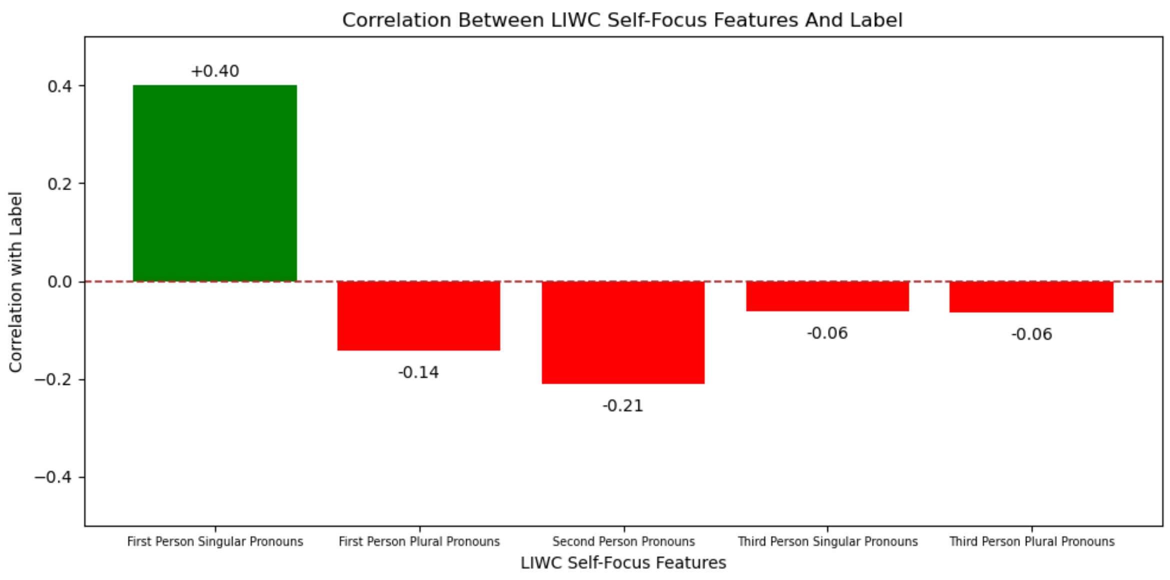
First Person Singular Pronouns stand out with the highest values overall, showing medians around 7 to 10 and upper quartiles above 10, which indicates that self-referential language is the most prominent pronoun type in both text categories. In contrast, **First Person Plural**, **Second Person** and **Third Person Plural Pronouns** all have medians at zero and very low quartiles, suggesting that collective or other-referential pronouns are used far less frequently. This distribution suggests that self-focus dominates pronoun use while references to others or groups are comparatively rare.

Differences between stressful and non-stressful texts:

Stressful texts consistently show higher values for First Person Singular Pronouns compared to non-stressful texts. This indicates that stressful texts are more self-focused, reflecting heightened personal involvement or introspection. First Person Plural and Second Person Pronouns show little difference between categories, suggesting that stress does not significantly alter collective or direct-address language. For Third Person Singular and Plural Pronouns, non-stressful texts show slightly higher values, implying that references to others may be more common in non-stressful contexts, while stressful texts remain more inwardly focused.

Now, we will explore the correlations between each subcategory within the LIWC group of self-focus features and the label.

In [63]:
```
self_focus_features = ['First Person Singular Pronouns', 'First Person Plural Pr
                       'Third Person Plural Pronouns']
correlations = dreaddit_train[self_focus_features + ['Label']].corr()['Label'].c
colors = ['green' if val >= 0 else 'red' for val in correlations]
plt.figure(figsize=(10, 5))
bars = plt.bar(correlations.index, correlations.values, color=colors)
for bar, val in zip(bars, correlations.values):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.01 if val >
             f"{val:+.2f}", ha='center', va='bottom' if val >= 0 else 'top', for

plt.axhline(0, color='brown', linewidth=1.2, linestyle='--')
plt.xticks(fontsize=7)
plt.ylabel("Correlation with Label")
plt.xlabel("LIWC Self-Focus Features")
plt.title("Correlation Between LIWC Self-Focus Features And Label")
plt.ylim(-0.5, 0.5)
plt.tight_layout()
plt.show()
```



Key Observations:

First Person Singular Pronouns show the strongest positive correlation (+0.40), indicating that self-referential language is highly associated with stressful texts. This suggests that when individuals are stressed, they tend to focus more on themselves and their personal experiences. In contrast, all other pronoun categories: **First Person Plural**

(-0.14), **Second Person** (-0.21), **Third Person Singular** (-0.06) and **Third Person Plural** (-0.06) show negative correlations. This means that collective pronouns, direct address pronouns and references to others are less likely to appear in stressful texts compared to non-stressful ones. The strongest negative correlation is for **Second Person Pronouns**, implying that stressful texts are particularly less oriented toward addressing others directly.

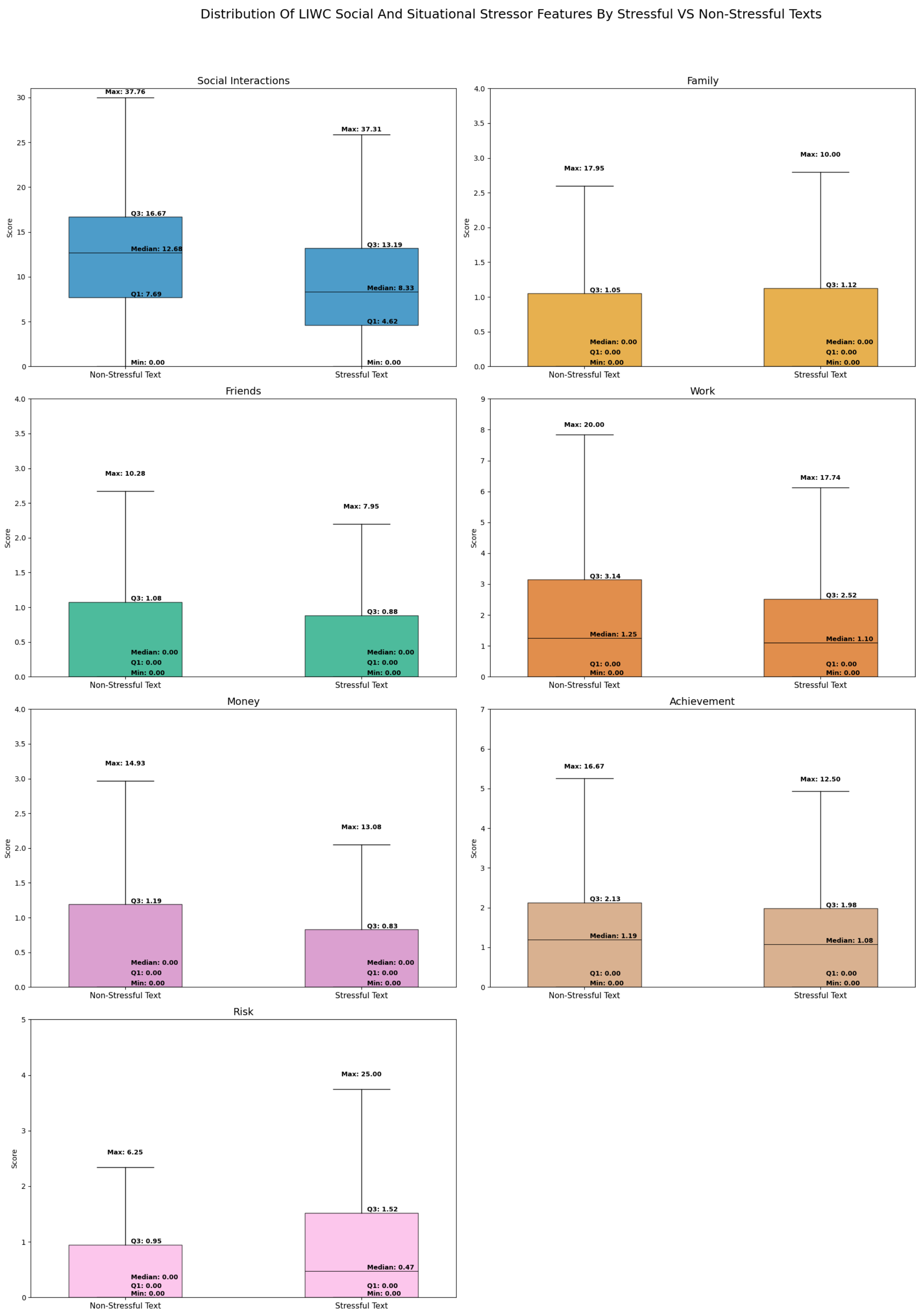
Possible Insights:

The overall pattern suggests that stressful texts are characterized by heightened self-focus and reduced outward orientation. The moderately-strong positive correlation for First Person Singular Pronouns highlights how stress drives individuals to center their language on themselves, reflecting personal involvement, introspection or self-concern. Meanwhile, the negative correlations for plural and other referential pronouns suggest that non-stressful texts are more likely to include collective perspectives or references to others, indicating a broader social orientation.

Visualization 7 Part 4: LIWC Social And Situational Stressors

Lastly, we will create boxplots to visualize the distribution of LIWC scores across stressful (label = 1) and non-stressful (label = 0) texts for the LIWC Social And Situational Stressors, which include: **Social Interactions**, **Family**, **Friends**, **Work**, **Money**, **Achievement** and **Risk**.

In [65]:
```python
# LIWC Social & Situational Stressor Features (already named as columns)
social_situational_features = ['Social Interactions', 'Family', 'Friends', 'Work
colors = sns.color_palette("colorblind", len(social_situational_features))
fig, axes = plt.subplots(4, 2, figsize=(20, 26))
axes = axes.flatten()

for i, feature in enumerate(social_situational_features):
    ax = axes[i]
    data = [dreaddit_train[dreaddit_train['Label'] == 0][feature].dropna(), drea
    positions = [1, 2.25]
    bp = ax.boxplot(data, positions=positions, widths=0.6, patch_artist=True, sh
    for patch in bp['boxes']:
        patch.set_facecolor(colors[i])
        patch.set_edgecolor("black")
        patch.set_alpha(0.7)
    whisker_tops = {}
    true_max_vals = {}
    for j, vals in enumerate(data):
        stats = {"Min": np.min(vals), "Q1": np.percentile(vals, 25), "Median": r
        true_max_vals[j] = np.max(vals)
        whisker_val = np.max([w.get_ydata().max() for w in bp['whiskers'] if w.g
        whisker_tops[j] = whisker_val
        stat_order = ["Min", "Q1", "Median", "Q3"]
        value_groups = {}
        for stat in stat_order:
            val = stats[stat]
            value_groups.setdefault(val, []).append(stat)
        for val, stat_list in value_groups.items():
            for idx, stat_name in enumerate(stat_list):
                offset = 0.15 * idx
                if feature == "Work":
                    if stat_name == "Q1" and stats["Q1"] == stats["Min"]:
                        offset += 0.15
                if feature == "Achievement":
                    if stat_name == "Q1" and stats["Q1"] == stats["Min"]:
                        offset += 0.10
                x_pos = positions[j] + 0.03
                ax.text(x_pos, val + offset, f"{stat_name}: {val:.2f}", ha='left
        if stats["Q3"] == 0.00:
            stacked_count = len(value_groups.get(0.00, []))
            max_y = stats["Q3"] + (0.15 * stacked_count)
            max_x = positions[j] + 0.03
            ha = 'left'
        else:
            max_y = whisker_tops[j] + 0.2
            max_x = positions[j]
            ha = 'center'
        ax.text(max_x, max_y, f"Max: {true_max_vals[j]:.2f}", ha=ha, va='bottom'
    ax.set_xticks([1, 2.25])
    ax.set_xticklabels(["Non-Stressful Text", "Stressful Text"], fontsize=11)
    ax.set_title(feature, fontsize=14)
    ax.set_ylabel("Score")
    ax.set_ylim(0, np.ceil(max(whisker_tops.values()) + 1))

for k in range(len(social_situational_features), len(axes)):
    axes[k].axis('off')
fig.suptitle("Distribution Of LIWC Social And Situational Stressor Features By S
plt.subplots_adjust(hspace=0.5)
plt.tight_layout(rect=[0, 0, 0.9, 0.95])
plt.show()
```



Key Observations:

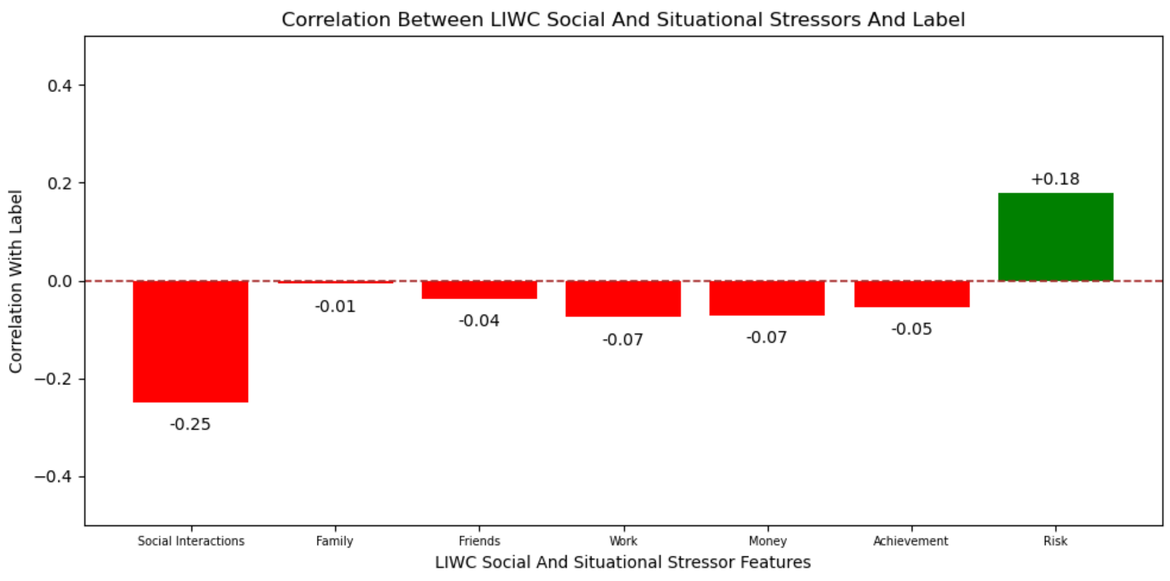
Social Interactions dominate the distributions, with medians around 8-13 and upper quartiles reaching 16-17, far higher than any other category. This suggests that interpersonal dynamics are the most linguistically salient domain across both stressful and non-stressful texts. **Work** and **Achievement** also show moderate values, with medians around 1-1.2 and quartiles near 2-3, indicating that professional and goal-related contexts are regularly mentioned. **Family**, **Friends**, **Money** and **Risk** generally have much lower medians, reflecting that these domains are less consistently present in texts. Risk is particularly interesting as while its median remains near zero, stressful texts show a much higher maximum, suggesting occasional spikes in risk-related language under stress.

Differences between stressful and non-stressful texts:

Non-stressful texts consistently show higher values in Social Interactions, implying that non-stressful contexts may involve richer descriptions of social engagement. Stressful texts however show slightly elevated values in Family and Risk, suggesting that stress is more often tied to family concerns and perceived risks. Work and Achievement show only minor differences, with non-stressful texts slightly higher, indicating that professional and goal-related language is not strongly differentiated by stress. Money and Friends are low in both categories, but non-stressful texts again show slightly higher quartiles, hinting that financial and friendship references are more common in non-stressful contexts.

Now, we will explore the correlations between each subcategory within the LIWC group of social and situational stressors and the label.

```
In [67]: social_stressor_features = ['Social Interactions', 'Family', 'Friends', 'Work','
         correlations = dreaddit_train[social_stressor_features + ['Label']].corr()['Labe
         colors = ['green' if val >= 0 else 'red' for val in correlations]
         plt.figure(figsize=(10, 5))
         bars = plt.bar(correlations.index, correlations.values, color=colors)
         for bar, val in zip(bars, correlations.values):
             plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.01 if val >
                      f"{val:+.2f}", ha='center', va='bottom' if val >= 0 else 'top', for

         plt.axhline(0, color='brown', linewidth=1.2, linestyle='--')
         plt.xticks(fontsize=7)
         plt.ylabel("Correlation With Label")
         plt.xlabel("LIWC Social And Situational Stressor Features")
         plt.title("Correlation Between LIWC Social And Situational Stressors And Label")
         plt.ylim(-0.5, 0.5)
         plt.tight_layout()
         plt.show()
```



Key Observations:

Social Interactions show the strongest negative correlation (-0.25), indicating that texts rich in social engagement language are more likely to be non-stressful. This suggests that references to socializing, relationships or general interaction tend to align with calmer contexts. **Family**, **Friends**, **Work**, **Money** and **Achievement** all display weak negative correlations (between -0.01 and -0.07), meaning they are slightly more common in non-stressful texts but not strongly predictive. **Risk** stands out as the only feature with a positive correlation (+0.18), showing that language related to danger, uncertainty, or exposure is more strongly associated with stressful texts.

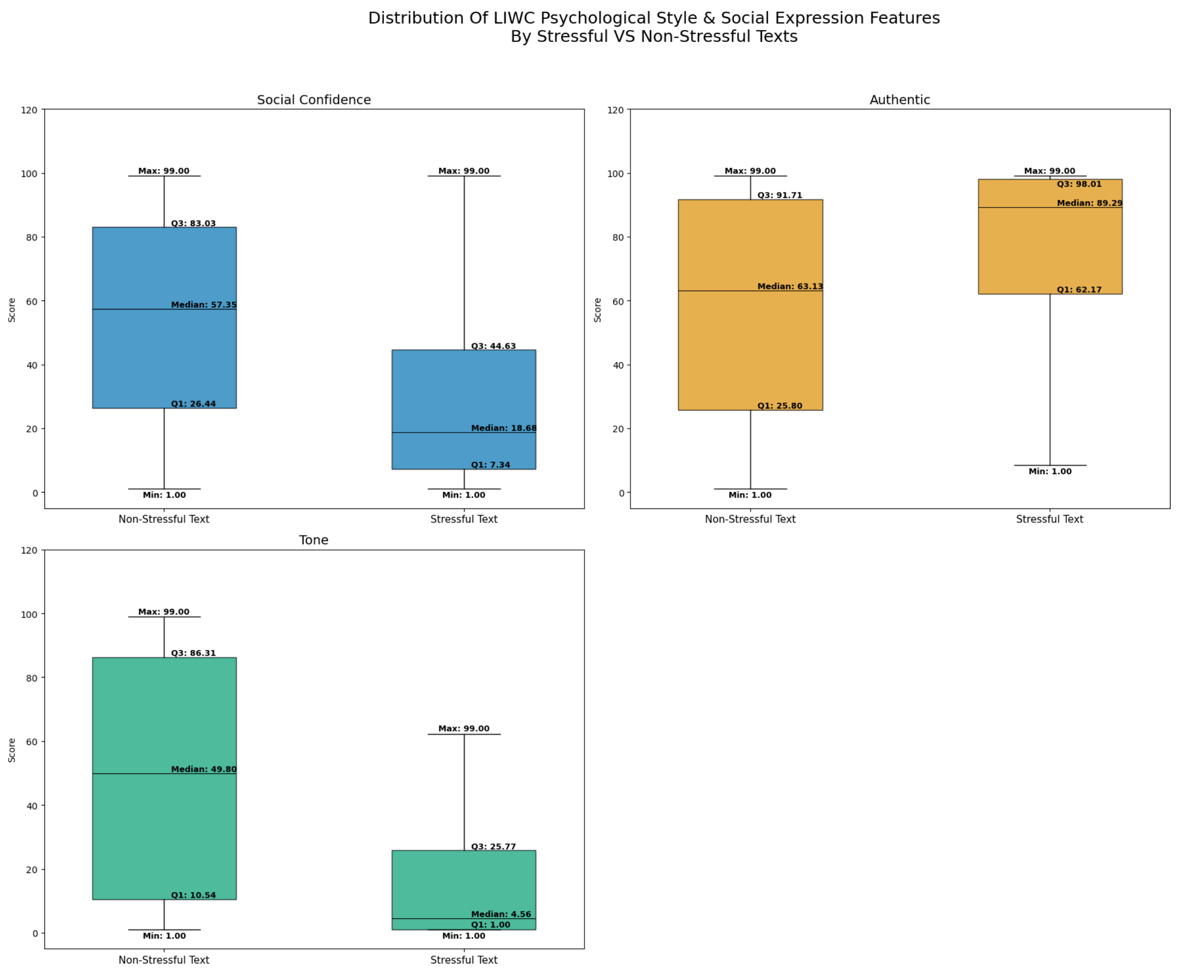
Possible Insights:

The overall pattern suggests that non-stressful texts emphasize social and situational domains such as interactions, work and achievement, albeit modestly. Stressful texts are distinguished by their stronger association with risk-related language. This makes sense conceptually: stress often arises when individuals perceive threats or uncertainties and this is reflected linguistically through references to risk. Meanwhile, the negative correlation for social interactions implies that engaging with others or describing social contexts may act as a buffer against stress, or at least be more characteristic of non-stressful narratives.

Visualization 7 Part 5: LIWC Psychological Style And Social Expression

Next, we will create boxplots to visualize the distribution of LIWC scores across stressful (label = 1) and non-stressful (label = 0) texts for the LIWC Psychological Style And Social Expression Features, which include: **Social Confidence**, **Authenticity** and **Tone**.

In [69]:
```python
# LIWC Psychological Style & Social Expression Features
psych_style_features = ['Social Confidence', 'Authentic', 'Tone']
colors = sns.color_palette("colorblind", len(psych_style_features))
fig, axes = plt.subplots(2, 2, figsize=(20, 15))
axes = axes.flatten()

for i, feature in enumerate(psych_style_features):
    ax = axes[i]
    data = [dreaddit_train[dreaddit_train['Label'] == 0][feature].dropna(), drea
    positions = [1, 2.25]
    bp = ax.boxplot(data, positions=positions, widths=0.6, patch_artist=True, sh
    for patch in bp['boxes']:
        patch.set_facecolor(colors[i])
        patch.set_edgecolor("black")
        patch.set_alpha(0.7)
    whisker_tops = {}
    whisker_bottoms = {}
    true_max_vals = {}
    for j, vals in enumerate(data):
        stats = {"Min": np.min(vals), "Q1": np.percentile(vals, 25), "Median": r
        true_max_vals[j] = np.max(vals)
        whiskers = [ w for w in bp['whiskers'] if w.get_xdata().mean() == positi
        whisker_tops[j] = max(w.get_ydata().max() for w in whiskers)
        whisker_bottoms[j] = min(w.get_ydata().min() for w in whiskers)
        stat_order = ["Min", "Q1", "Median", "Q3"]
        value_groups = {}
        for stat in stat_order:
            value_groups.setdefault(stats[stat], []).append(stat)
        for val, stat_list in value_groups.items():
            for idx, stat_name in enumerate(stat_list):
                offset = 0.15 * idx
                x_pos = positions[j] + 0.03
                if stat_name == "Min":
                    ax.text(positions[j], whisker_bottoms[j] - 0.5, f"Min: {val:
                            fontweight="bold")
                    continue
                if feature == "Authentic" and j == 1 and stat_name == "Q3":
                    offset -= 2.7
                if feature == "Tone" and j == 1:
                    if stats["Min"] == stats["Q1"] == 1.00:
                        if stat_name == "Q1":
                            offset += 0.15
                ax.text(x_pos, val + offset, f"{stat_name}: {val:.2f}", ha='left
        if stats["Q3"] == 0.00:
            stacked_count = len(value_groups.get(0.00, []))
            max_y = stats["Q3"] + (0.15 * stacked_count)
            max_x = positions[j] + 0.03
            ha = 'left'
        else:
            max_y = whisker_tops[j] + 0.3
            max_x = positions[j]
            ha = 'center'
        ax.text(max_x, max_y, f"Max: {true_max_vals[j]:.2f}", ha=ha, va='bottom'
    ax.set_xticks([1, 2.25])
    ax.set_xticklabels(["Non-Stressful Text", "Stressful Text"], fontsize=11)
    ax.set_title(feature, fontsize=14)
    ax.set_ylabel("Score")
    ax.set_ylim(-5, 120)

for k in range(len(psych_style_features), len(axes)):
    axes[k].axis('off')
fig.suptitle("Distribution Of LIWC Psychological Style & Social Expression Featu
plt.subplots_adjust(hspace=0.5)
plt.tight_layout(rect=[0, 0, 0.9, 0.95])
plt.show()
```



Key Observations:

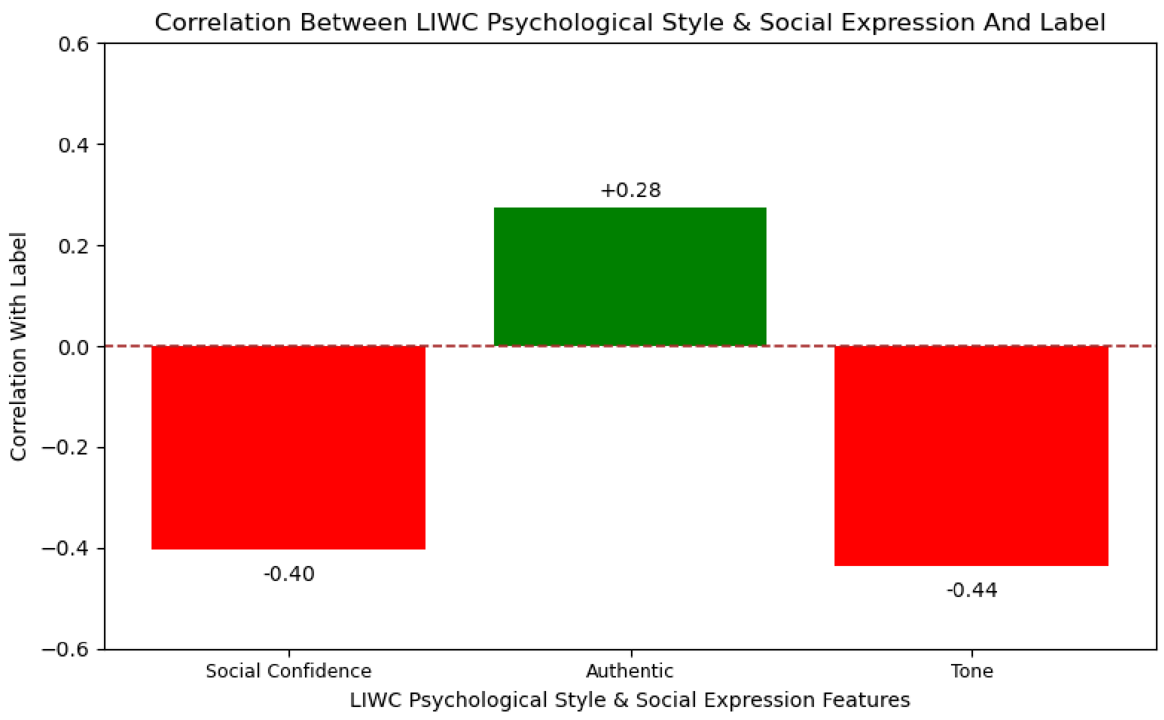
Social Confidence and **Tone** show the widest separation between the two text types. Non-stressful texts have much higher medians (57.35 for Social Confidence and 49.80 for Tone) compared to stressful texts (18.68 and 4.56 respectively). This suggests that non-stressful texts are characterized by more confident and positive language, while stressful texts lean toward uncertainty and negativity. **Authenticity** shows the opposite pattern: stressful texts have a much higher median (89.22) compared to non-stressful texts (63.13). This indicates that stressful texts are more likely to use language perceived as genuine or self-disclosing, whereas non-stressful texts balance authenticity with more socially polished expression.

Differences between stressful and non-stressful texts:

Non-stressful texts consistently demonstrate higher values in Social Confidence and Tone, reflecting a more positive, assured and socially engaging style. Stressful texts in contrast show lower confidence and tone scores, aligning with the idea that stress reduces positivity and increases linguistic markers of insecurity. The sharp rise in Authenticity for stressful texts suggests that individuals under stress may express themselves in a more raw, unfiltered manner, prioritizing honesty over social presentation. This divergence between authenticity and confidence highlights a key psychological dynamic: stress fosters genuine self-expression but diminishes positivity and social assurance.

Now, we will explore the correlations between each subcategory within the LIWC Psychological Style & Social Expression Features and the label.

In [71]:
```
psych_style_features = ['Social Confidence', 'Authentic', 'Tone']
correlations = dreaddit_train[psych_style_features + ['Label']].corr()['Label'].
colors = ['green' if val >= 0 else 'red' for val in correlations]
plt.figure(figsize=(8, 5))
bars = plt.bar(correlations.index, correlations.values, color=colors)
for bar, val in zip(bars, correlations.values):
    plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height() + 0.01 if val >
             f"{val:+.2f}", ha='center', va='bottom' if val >= 0 else 'top', for

plt.axhline(0, color='brown', linewidth=1.2, linestyle='--')
plt.xticks(fontsize=9)
plt.ylabel("Correlation With Label")
plt.xlabel("LIWC Psychological Style & Social Expression Features")
plt.title("Correlation Between LIWC Psychological Style & Social Expression And
plt.ylim(-0.6, 0.6)
plt.tight_layout()
plt.show()
```



Key Observations:

Tone shows the strongest negative correlation (-0.44), meaning that texts with more positive or upbeat tone are much less likely to be stressful. **Social Confidence** follows closely with another strong negative correlation (-0.40), suggesting that confident, assured language is also more characteristic of non-stressful texts. In contrast, **Authenticity** stands out with a positive correlation (+0.28), indicating that stressful texts are more likely to contain language perceived as genuine, raw or self-disclosing.

Possible Insights:

Non-stressful texts are marked by higher confidence and more positive tone, reflecting a socially polished and optimistic style. Stressful texts, however, show the opposite pattern:

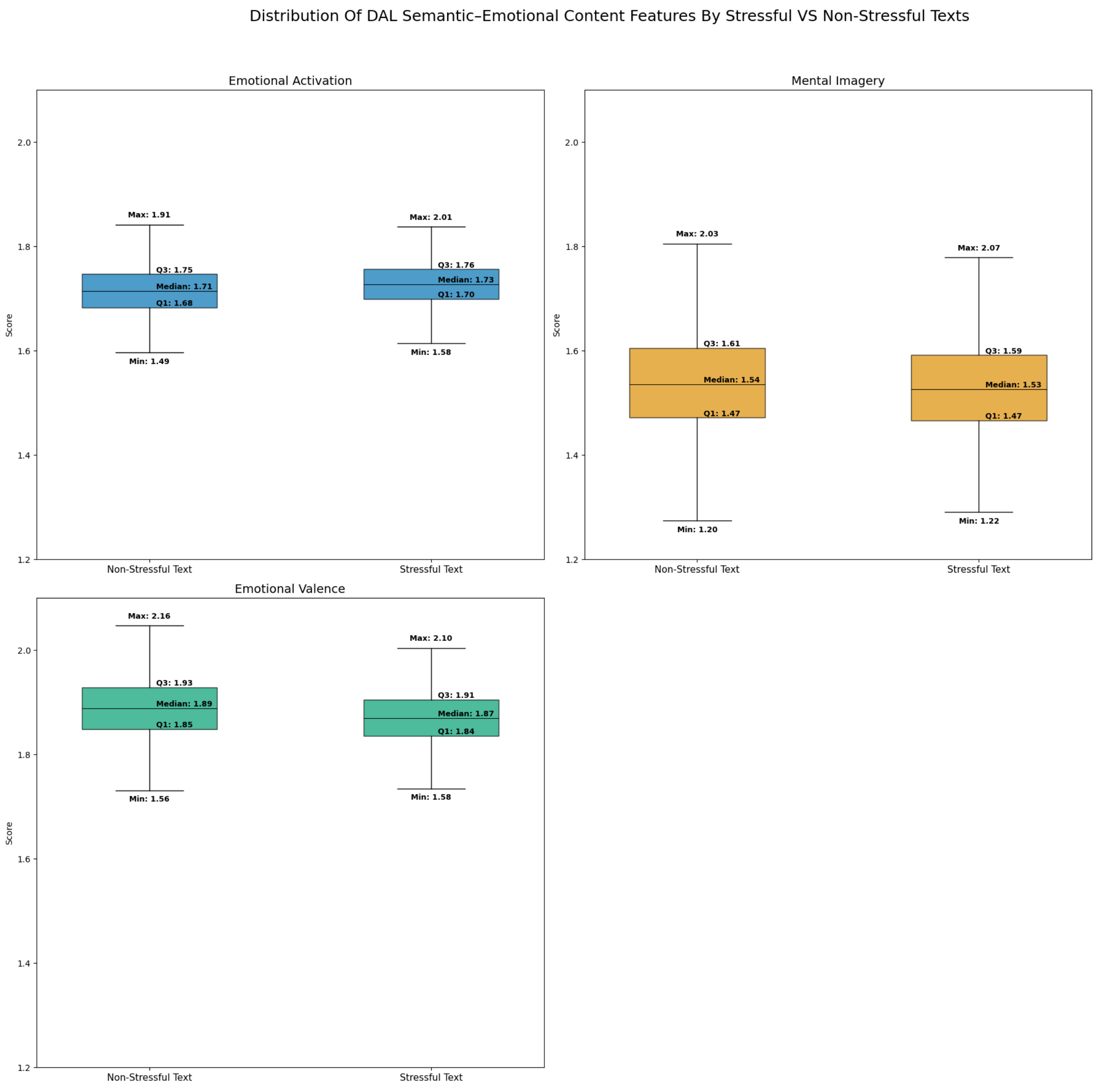
lower confidence and tone scores but higher authenticity. This suggests that under stress, individuals may abandon socially polished language and instead express themselves in a more direct, unfiltered way. The negative correlations for confidence and tone highlight how stress diminishes social assurance and positivity, while the positive correlation for authenticity reveals that stress fosters genuine self-expression. Stressful texts are characterized by authenticity but lack confidence and positivity, while non-stressful texts emphasize social assurance and optimistic tone.

Visualization 7 Part 6: DAL Semantic-Emotional Content

Lastly, we will create boxplots to visualize the distribution of LIWC scores across stressful (label = 1) and non-stressful (label = 0) texts for the DAL Semantic-Emotional Content, which include: **Emotional Activation**, **Mental Imagery** and **Emotional Valence**.

In [73]:
```
# DAL Semantic-Emotional Content Features
dal_features = ['Emotional Activation', 'Mental Imagery', 'Emotional Valence']
colors = sns.color_palette("colorblind", len(dal_features))
fig, axes = plt.subplots(2, 2, figsize=(20, 18))
axes = axes.flatten()
for i, feature in enumerate(dal_features):
    ax = axes[i]
    data = [dreaddit_train[dreaddit_train['Label'] == 0][feature].dropna(), drea
    positions = [1, 2.25]
    bp = ax.boxplot(data, positions=positions, widths=0.6, patch_artist=True, sh
    for patch in bp['boxes']:
        patch.set_facecolor(colors[i])
        patch.set_edgecolor("black")
        patch.set_alpha(0.7)
    whisker_tops = {}
    whisker_bottoms = {}
    true_max_vals = {}
    for j, vals in enumerate(data):
        stats = {"Min": np.min(vals), "Q1": np.percentile(vals, 25), "Median": r
        true_max_vals[j] = np.max(vals)
        whisker_vals = [w.get_ydata() for w in bp['whiskers'] if w.get_xdata().m
        whisker_tops[j] = np.max([w.max() for w in whisker_vals])
        whisker_bottoms[j] = np.min([w.min() for w in whisker_vals])
        stat_order = ["Q1", "Median", "Q3"]
        value_groups = {}
        for stat in stat_order:
            val = stats[stat]
            value_groups.setdefault(val, []).append(stat)
        for val, stat_list in value_groups.items():
            for idx, stat_name in enumerate(stat_list):
                offset = 0.15 * idx
                x_pos = positions[j] + 0.03
                ax.text(x_pos, val + offset, f"{stat_name}: {val:.2f}", ha='left
        ax.text(positions[j], whisker_bottoms[j] - 0.01, f"Min: {stats['Min']:.2
                fontweight="bold")
        ax.text(positions[j], whisker_tops[j] + 0.01, f"Max: {true_max_vals[j]:.
                fontweight="bold")
    ax.set_xticks([1, 2.25])
    ax.set_xticklabels(["Non-Stressful Text", "Stressful Text"], fontsize=11)
    ax.set_title(feature, fontsize=14)
    ax.set_ylabel("Score")
    ax.set_ylim(1.2, 2.1)
```

```
for k in range(len(dal_features), len(axes)):
    axes[k].axis('off')

fig.suptitle("Distribution Of DAL Semantic–Emotional Content Features By Stressf
plt.subplots_adjust(hspace=0.5)
plt.tight_layout(rect=[0, 0, 0.9, 0.95])
plt.show()
```



Key Observations:

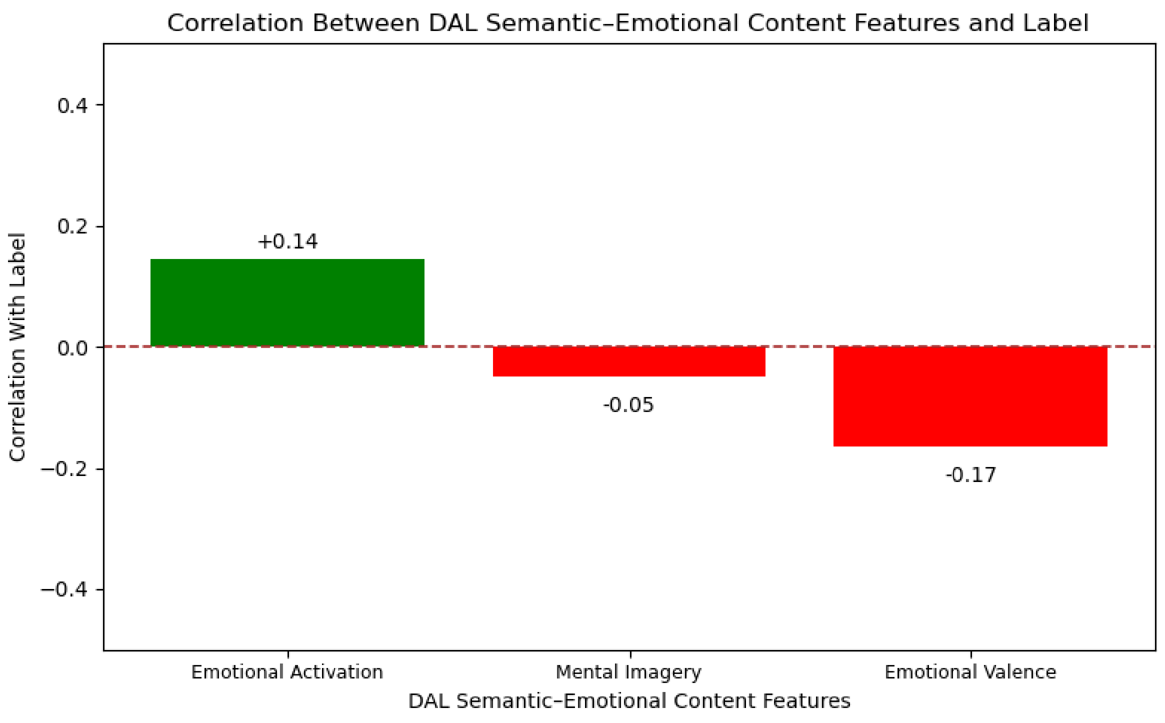
For **Emotional Activation** the medians score is around 1.71 to 1.73 and upper quartiles near 1.75 to 1.76, suggesting that both stressful and non-stressful texts are consistently marked by moderate emotional arousal. **Mental Imagery** has median score of around 1.53 to 1.54 and quartiles near 1.59 to 1.61, indicating that descriptive, image-evoking language is moderately present in both categories. **Emotional Valence** is slightly higher than the other two features, with medians around 1.87 to 1.89 and quartiles near 1.91 to 1.93, reflecting a tendency toward more positive or pleasant emotional tone overall. The ranges across all three features are relatively narrow, showing that these semantic-emotional dimensions are fairly stable across texts.

Differences between stressful and non-stressful texts:

Stressful texts show slightly higher Emotional Activation, suggesting that stress is associated with marginally more emotionally charged language. Mental Imagery is nearly identical between the two categories, indicating that stress does not significantly alter the use of descriptive or image-based language. Emotional Valence is slightly lower in stressful texts, suggesting that stressful texts lean toward less positive emotional tone compared to non-stressful ones.

Now, we will explore the correlations between each subcategory within the DAL Semantic-Emotional Content and the label.

In [75]:
```python
dal_features = ['Emotional Activation', 'Mental Imagery', 'Emotional Valence']
correlations = dreaddit_train[dal_features + ['Label']].corr()['Label'].drop('La
colors = ['green' if val >= 0 else 'red' for val in correlations]
plt.figure(figsize=(8, 5))
bars = plt.bar(correlations.index, correlations.values, color=colors)
for bar, val in zip(bars, correlations.values):
    y_pos = bar.get_height() + 0.01 if val >= 0 else bar.get_height() - 0.03
    plt.text(bar.get_x() + bar.get_width() / 2, y_pos, f"{val:+.2f}", ha='center

plt.axhline(0, color='brown', linewidth=1.2, linestyle='--')
plt.xticks(fontsize=9)
plt.ylabel("Correlation With Label")
plt.xlabel("DAL Semantic-Emotional Content Features")
plt.title("Correlation Between DAL Semantic-Emotional Content Features and Label
plt.ylim(-0.5, 0.5)
plt.tight_layout()
plt.show()
```



Key Observations:

Emotional Activation shows a positive correlation (+0.14), suggesting that texts with slightly higher emotional intensity are more likely to be stressful. **Mental Imagery**, however, has a weak negative correlation (-0.05), indicating that descriptive, image-evoking language is slightly more common in non-stressful texts. **Emotional Valence**

stands out with a stronger negative correlation (-0.17), meaning that texts with more positive emotional tone are less likely to be stressful.

Possible Insights:

Stressful texts are characterized by greater emotional activation, reflecting more charged or intense language. At the same time, they show lower emotional valence, aligning with the idea that stress reduces positivity and shifts language toward a more negative or neutral tone. The slight negative correlation for mental imagery suggests that non-stressful texts may include richer descriptive detail, while stressful texts focus more on emotional intensity rather than vivid imagery.

Visualization 7 Part 7: Visualizing Factors With Strongest Positive And Negative Correlations

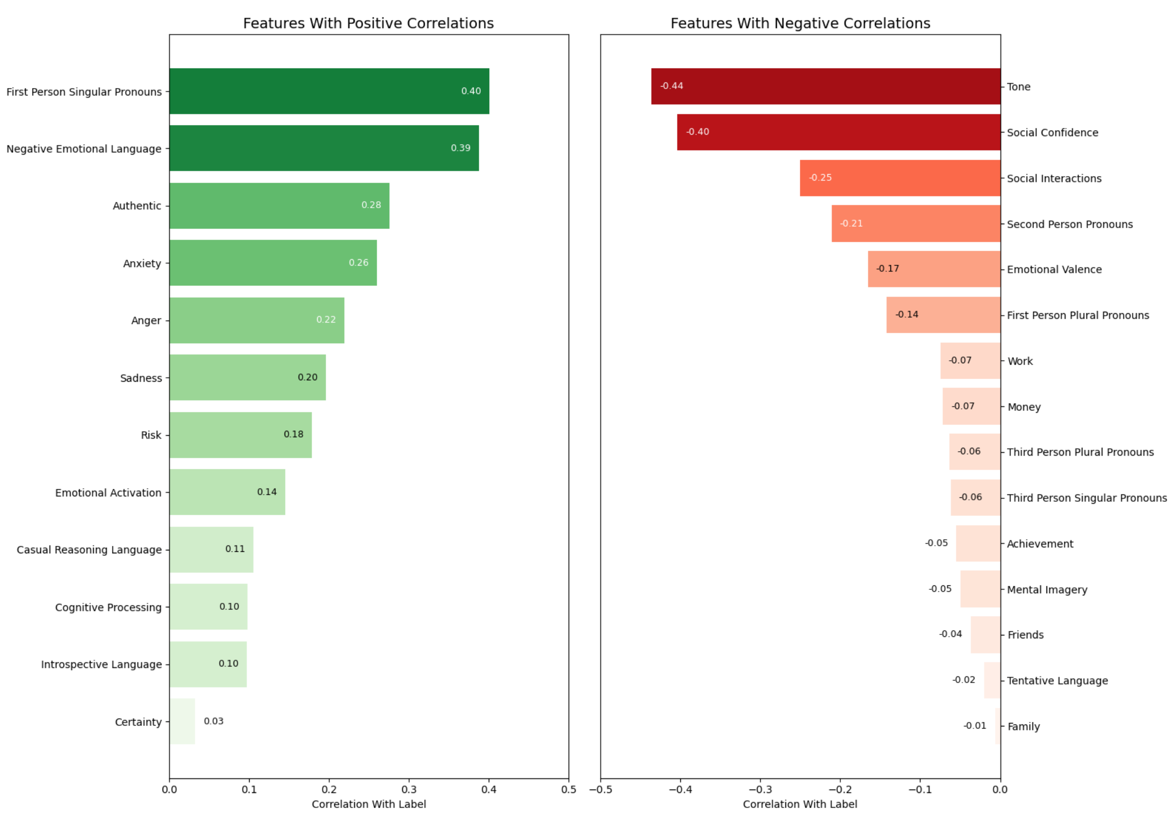
The purpose of this visualization is to identify which linguistic and psychological features are most strongly associated with stress in the Dreaddit dataset. By isolating features with the strongest positive correlations, we can determine which factors increase the likelihood of a text being classified as stressful. Conversely, features with the strongest negative correlations highlight those most associated with non-stressful text. This comparison allows researchers to quickly focus on the most informative features, facilitating targeted analysis, intervention or feature selection for downstream modeling tasks.

The plot is a side-by-side horizontal bar chart displaying the features with the strongest positive and negative correlations with the stress label. The left panel shows features whose higher values are associated with stressful texts (positive correlations), while the right panel shows features whose higher values are associated with non-stressful texts (negative correlations). By examining the length and color intensity of each bar, one can quickly identify which features are most influential. Longer bars indicate stronger correlations, and darker colors represent stronger effect sizes, allowing for visual prioritization of key factors for further analysis or modeling.

In [77]:
```
# Combine all relevant LIWC features (affective, cognitive, self-focus, social s
all_features = ['Negative Emotional Language', 'Anxiety', 'Sadness', 'Anger', 'C
                'Casual Reasoning Language', 'Tentative Language', 'Certainty', 
                'Second Person Pronouns', 'Third Person Singular Pronouns', 'Thi
                'Work', 'Money', 'Achievement', 'Risk', 'Social Confidence', 'Au
                'Mental Imagery', 'Emotional Valence']
correlations = dreaddit_train[all_features + ['Label']].corr()['Label'].drop('La
pos_corr = correlations[correlations > 0].sort_values(ascending=False)
neg_corr = correlations[correlations < 0].sort_values()
pos_cmap = plt.cm.Greens
neg_cmap = plt.cm.Reds

def get_color(val, cmap, max_val):
    norm_val = val / max_val if max_val != 0 else 0
    return cmap(norm_val)
fig, axes = plt.subplots(1, 2, figsize=(16, 12))

# Positive correlations
max_pos = 0.5
```

```python
for i, (feat, val) in enumerate(pos_corr.items()):
    axes[0].barh(feat, val, color=get_color(val, pos_cmap, max_pos))
    if abs(val) > 0.20:
        text_color = 'white'
        ha = 'right'
        x_text = val - 0.01
    elif abs(val) < 0.035:
        text_color = 'black'
        ha = 'left'
        x_text = val + 0.01
    else:
        text_color = 'black'
        ha = 'right'
        x_text = val - 0.01
    axes[0].text(x_text, i, f"{val:.2f}", va='center', ha=ha, fontsize=9, color=
axes[0].set_title("Features With Positive Correlations", fontsize=14)
axes[0].set_xlabel("Correlation With Label")
axes[0].set_xlim(0, 0.5)  # set x-axis limit
axes[0].invert_yaxis()

# Negative correlations
max_neg = 0.5  # fixed limit
for i, (feat, val) in enumerate(neg_corr.items()):
    axes[1].barh(feat, val, color=get_color(abs(val), neg_cmap, max_neg))
    if abs(val) > 0.20:
        text_color = 'white'
        ha = 'left'
        x_text = val + 0.01
    elif abs(val) < 0.055:
        text_color = 'black'
        ha = 'right'
        x_text = val - 0.01
    else:
        text_color = 'black'
        ha = 'left'
        x_text = val + 0.01
    axes[1].text(x_text, i, f"{val:.2f}", va='center', ha=ha, fontsize=9, color=

axes[1].set_title("Features With Negative Correlations", fontsize=14)
axes[1].set_xlabel("Correlation With Label")
axes[1].set_xlim(-0.5, 0)  # set x-axis limit
axes[1].invert_yaxis()
axes[1].yaxis.tick_right()
axes[1].yaxis.set_label_position("right")
plt.suptitle("LIWC Features With Positive And Negative Correlations With Stress
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```

LIWC Features With Positive And Negative Correlations With Stress Label



Predictive Modelling

Problem Outline:

The primary goal of this project is to predict the stress label of text submissions in a social platform dataset (Reddit) using both textual and psycholinguistic features. Each text entry in the dataset is labeled as either stressful (Label = 1) or non-stressful (Label = 0), reflecting whether the author is expressing stress-related content. Accurate prediction of these labels is crucial for early identification of individuals experiencing high levels of stress, which can inform interventions in digital mental health platforms, academic monitoring systems or workplace wellness tools.

Columns In The Cleaned Datasets:

The dataset consists of 35 columns, which can be broadly categorized into three groups:

1. **Textual Data:** The Text column contains the raw written content of each post or sentence. This is unstructured data, and its meaning and sentiment need to be captured using natural language processing (NLP) techniques.

2. **Psycholinguistic And Affective Features:** These include LIWC features such as Affective (Negative Emotional Language, Anxiety, Sadness, Anger), Cognitive (Cognitive Processing, Certainty), Self-focus (first/third person pronouns), Social/psychological style (Authentic, Tone, Social Confidence), and Social Stressor features (Family, Work, Achievement). Additionally, DAL (Dictionary of Affect in Language) Semantic–Emotional Content features capture emotional activation,

mental imagery, and emotional valence of the words used in the texts. These numeric features provide structured indicators of emotional and cognitive states and can complement text embeddings for machine learning models.

3. **Meta Or Social Engagement Features:** Columns like Upvote Proportion and Number of Comments provide context about audience engagement, which may correlate with stress expression or social support.

Challenges We Will Face During Modelling Process:

1. **Class Imbalance:** The dataset is slightly imbalanced, with significantly more stressful texts than non-stressful ones. This can cause naive models to favor the majority class, leading to poor detection of non-stressful texts. Metrics like accuracy may be misleading, hence F1-score (the one we are using), precision and recall are better indicators of model performance.

2. **Heterogeneous Features:** The dataset contains mixed types of features - numeric psycholinguistic scores and unstructured text. Combining these effectively in a single model requires careful preprocessing and thoughtful model architecture.

3. **Complex Textual Semantics:** Stressful content may be subtle or implicit. Simple word-count methods may fail to capture context, negation, sarcasm or nuanced emotion. This motivates the use of advanced NLP approaches such as complex deep learning models which capture contextual meaning.

Proposed Methods for Predictive Modeling:

Given the complexity of the dataset and the nature of the problem, a multi-method approach is needed. We aim to compare and evaluate classical machine learning and deep learning models, leveraging both text and numeric features.

Method 1: Classical Machine Learning Techniques

Classical machine learning models provide a strong baseline and are effective when working with structured features like LIWC, DAL and social engagement metrics. The plan includes:

- **Logistic Regression:** A simple, interpretable model that predicts the probability of a text being stressful. It can provide feature importance insights from LIWC/DAL features. Class imbalance can be addressed via class weights or oversampling.

- **Random Forest / Gradient Boosting (XGBoost, LightGBM, CatBoost)**:

Ensemble methods that capture non-linear relationships between psycholinguistic features and stress labels. These models are robust to noisy features and can provide feature importance rankings. Oversampling or SMOTE can help with class imbalance.

- **Support Vector Machines (SVMs):** Particularly effective for high-dimensional feature spaces such as TF-IDF vectorized text. Kernel methods can capture non-linear separations between stressful and non-stressful texts.

Additionally, regarding text preprocessing for classical ML, we will be relying on **tokenization**, **lowercasing** and **stopword removal**. We will also be converting text to numerical representations using **TF-IDF vectors**.

Method 2: Deep Learning Techniques

Deep learning models can capture complex patterns in text, especially sequences of words and subtle emotional cues. Proposed models include:

- **Convolutional Neural Networks (CNNs) for Text:** CNNs can capture local n-gram features and patterns in textual stress indicators, such as repeated negative emotional words.
- **Recurrent Neural Networks (RNNs) / LSTMs / GRUs:** Designed to process sequential data, ideal for capturing context in sentences or short posts. They can encode patterns of emotional expression, self-focus and narrative flow in the text.
- **Bidirectional LSTMs**

Method 3: Transformer-Based NLP Models

Transformer models, such as BERT, RoBERTa and DistilBERT, have achieved state-of-the-art performance on text classification tasks by understanding contextual meaning We will fine-tune pre-trained transformer models on the Text column to directly classify posts as stressful or non-stressful, use tokenization consistent with the transformer architecture and combine transformer embeddings with LIWC/DAL features in a multimodal architecture by concatenating the CLS token embedding with numeric features and passing them through a dense layer.

Handling Class Imbalances:

The dataset is slightly imbalanced, so several strategies will be carried out during the modelling process:

- **Resampling methods:** Oversampling the minority class using SMOTE or random oversampling.
- **Class weights:** Assigning higher loss weights to stressful texts during model training.
- **Evaluation metrics:** Focus on F1-score, precision and recall rather than overall accuracy as they better reflect minority class performance (this aligns with our focus on F1 scores)

Evaluation and Validation Strategy:

To ensure reliable model performance, the plan includes:

- Splitting `dreaddit_train` into 80% training and 20% validation subsets for hyperparameter tuning.

- Using the `dreaddit_test` set only for final evaluation.
- Performing cross-validation where appropriate (5-fold CV for classical ML) to assess robustness.

Comparative Analysis of Methods:

The project will compare performance across methods:

1. Classical ML vs Deep Learning vs Transformer models.
2. Text-only models vs numeric features only vs combined multimodal models.
3. Aim to identify which combination of text representation, numeric features and model architecture produces the highest F1-score.

This multi-method approach ensures we leverage both the psycholinguistic richness of the numeric features and the semantic depth of text embeddings, providing a holistic predictive modeling framework for stress detection.

Method 1: Classical Machine Learning Methods

Method 1 Model 1: Logistic Regression

Logistic regression is a supervised classification model that estimates the probability of a binary outcome. In this case, whether a Reddit post is stressful or non-stressful by modeling a linear relationship between the input features and the log-odds of the target variable. Despite its simplicity, logistic regression is a strong baseline model, particularly when working with interpretable numeric features such as psycholinguistic, emotional and social indicators.

In this dataset, logistic regression is especially suitable because many features (LIWC affective, cognitive, self-focus, social stressors and DAL semantic-emotional scores) are continuous, normalized and theoretically grounded. These features allow the model to quantify how specific linguistic and psychological attributes contribute to stress expression. However, raw text-based fields (such as Text and Subcategory) cannot be directly used in logistic regression without prior transformation (TF-IDF or embeddings) and are therefore excluded at this stage. For **Sentence Range** and **Subcategory** columns, we will need to perform encoding on these features before fitting the model.

Logistic Regression Pipeline Process:

Step 1: Feature Selection And Encoding

Select all relevant non-textual features while excluding raw text fields and identifiers. For categorical contextual features, apply one-hot encoding, dropping one reference category to avoid multicollinearity. This allows the model to capture domain-specific stress patterns without imposing artificial ordering. For sentence range, encode as an ordinal numeric feature, reflecting increasing text length and expressive depth. Excluded columns include text. This step ensures the model operates on interpretable, structured inputs while incorporating both linguistic content and contextual signals.

Step 2: Feature Scaling

We will proceed to standardize all continuous numeric features using z-score normalization (StandardScaler). Logistic regression is sensitive to feature magnitude. Scaling ensures coefficients are comparable and regularization is applied fairly. One-hot encoded variables are not scaled, as they are already binary. This step improves numerical stability and coefficient interpretability.

```
In [80]: from sklearn.preprocessing import OneHotEncoder, StandardScaler
         from sklearn.compose import ColumnTransformer
         from sklearn.preprocessing import OrdinalEncoder

         # Target column (the variable we want to predict)
         target_col = 'Label'

         # List of numeric features (continuous variables to be standardized)
         numeric_features = ['Negative Emotional Language', 'Anxiety', 'Sadness', 'Anger'
                             'Casual Reasoning Language', 'Tentative Language', 'Certaint
                             'Second Person Pronouns', 'Third Person Singular Pronouns',
                             'Friends', 'Work', 'Money', 'Achievement', 'Risk', 'Social C
                             'Mental Imagery', 'Emotional Valence', 'Sentiment', 'Upvote

         # List of categorical features (nominal variables to be one-hot encoded)
         categorical_features = ['Subcategory']

         # List of ordinal features (ordered categories to be ordinal encoded)
         ordinal_features = ['Text Length']

         # Define the explicit order for the ordinal feature "Text Length"
         sentence_range_order = [['Very Short Text', 'Short Text', 'Medium Text', 'Long T

         # Separate predictors (X) and target (y) from training data
         X = dreaddit_train[numeric_features + categorical_features + ordinal_features]
         y = dreaddit_train[target_col]

         # Define preprocessing pipeline:
         # - OneHotEncoder for categorical features (drop first category to avoid colline
         # - OrdinalEncoder for ordered text length categories
         # - StandardScaler for numeric features (normalize to mean=0, std=1)
         preprocessor = ColumnTransformer(
             transformers=[('cat', OneHotEncoder(drop='first', handle_unknown='ignore'),
                           ('ord', OrdinalEncoder(categories=sentence_range_order), ordir
                           ('num', StandardScaler(), numeric_features)])
```

Step 3: Handling Class Imbalances

To address the class imbalance in the Dreaddit dataset, where non-stressful posts constitute a minority, **Synthetic Minority Over-sampling Technique (SMOTE)** is employed during model training. **SMOTE** is a resampling technique that synthetically generates new minority-class examples rather than simply duplicating existing ones. It works by selecting a minority-class instance and interpolating new samples along the line segments connecting it to its nearest minority neighbors in the feature space. By creating realistic, non-identical synthetic data points, SMOTE increases class balance while preserving the underlying data distribution.

However, before applying SMOTE, it is essential to perform a train-validation split on the original training dataset. This step is crucial to prevent information leakage, which occurs when knowledge from validation or test data inadvertently influences the training process. The dataset `dreaddit_train` will first be split into an **80% training set and a 20% validation set**, with the split stratified by the Label to preserve the original class distribution in both subsets. SMOTE is then applied only to the training portion, where the model learns decision boundaries.

In [82]:
```
from sklearn.model_selection import train_test_split

# Split the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_st

# Fit the preprocessor on training data and transform both train and validation
X_train_processed = preprocessor.fit_transform(X_train)
X_val_processed = preprocessor.transform(X_val)

from imblearn.over_sampling import SMOTE

# Initialize SMOTE (Synthetic Minority Oversampling Technique)
smote = SMOTE(sampling_strategy='auto', random_state=42)

# Apply SMOTE only on the training set (not validation/test)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_processed, y_t
```

Step 4: Model Training

At this stage, a regularized logistic regression model is trained on the processed and rebalanced training data. Logistic regression models the probability that a post is stressful by learning a linear combination of psycholinguistic, emotional, social and contextual features, followed by a sigmoid transformation. **L2 regularization** is applied to penalize large coefficients, reducing overfitting and improving generalization, especially given the relatively high dimensionality introduced by one-hot encoding of categorical variables.

The model is optimized using the **log-loss (cross-entropy) objective**, which directly aligns with probabilistic classification and allows meaningful threshold-based decisiess.

In [84]:
```
from sklearn.linear_model import LogisticRegression

# Initialize Logistic Regression model
log_reg = LogisticRegression(penalty='l2', solver='lbfgs', max_iter=5000, randon

# Train the model on resampled training data
log_reg.fit(X_train_resampled, y_train_resampled)

# Predict on validation set
y_val_pred = log_reg.predict(X_val_processed)
```

Step 5: Model Evaluation And Performance

The trained model is evaluated on the held-out validation set, which preserves the original class imbalance and represents real-world performance. Because stressful posts form the minority class and false negatives are particularly costly, the F1-score is used as the primary evaluation metric. The F1-score balances precision (avoiding false positives) and recall (capturing stressful posts), making it more informative than raw accuracy in imbalanced settings.

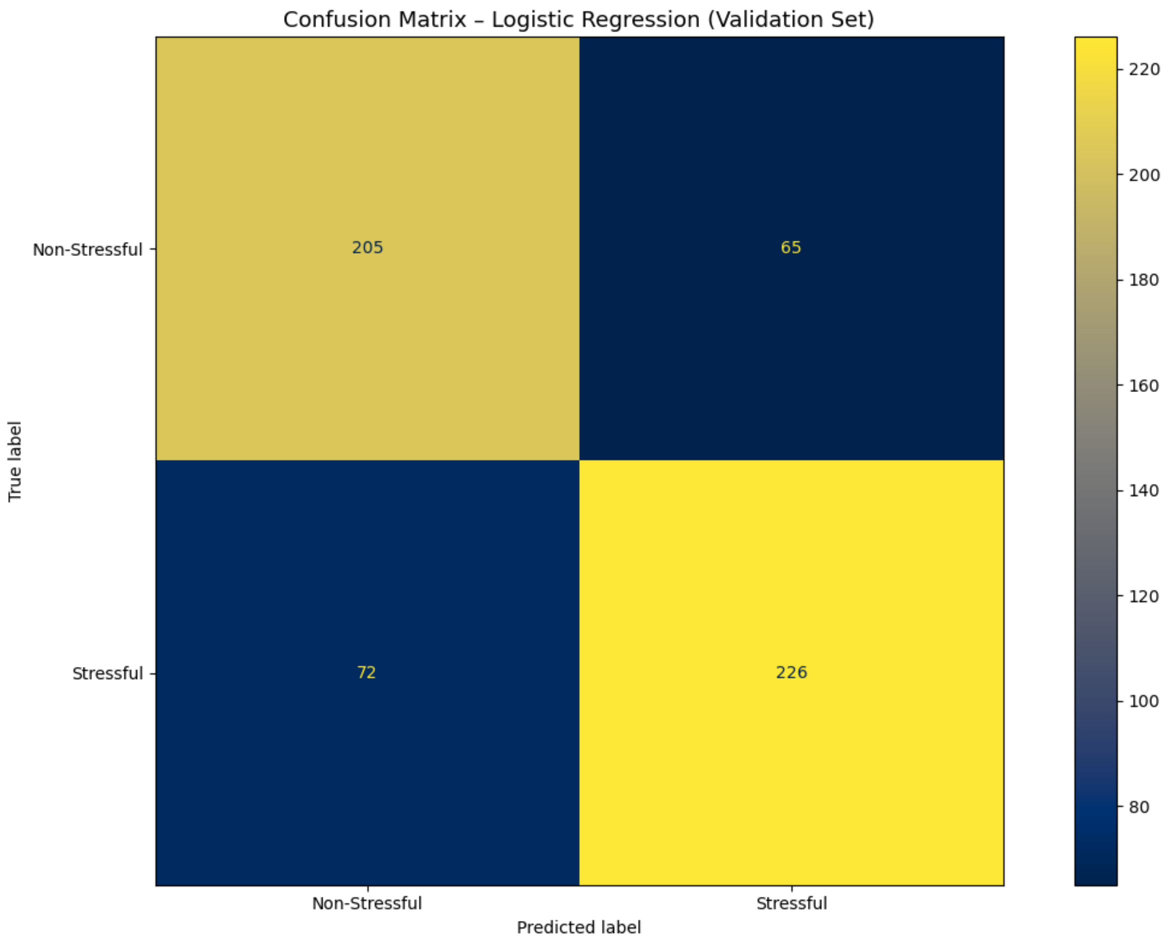
In [86]:
```
from sklearn.metrics import f1_score, precision_score, recall_score, confusion_m
f1 = f1_score(y_val, y_val_pred)
conf_matrix = confusion_matrix(y_val, y_val_pred)

print("Validation Performance (Logistic Regression)")
print("----------------------------------------------")
print(f"F1-score   : {f1:.4f}")
```

```
Validation Performance (Logistic Regression)
---------------------------------------------
F1-score   : 0.7674
```

After using the Logistic Regression Model, we **observe that the F1-Score is around 0.7674 (76% to 77%)**. This represents a strong and promising baseline result, especially given the highly imbalanced nature of the dataset and the complexity of stress-related language. For a linear model trained primarily on aggregated psycholinguistic, emotional and social features rather than raw text, this performance suggests that the engineered features capture meaningful stress-related signals and that the preprocessing, scaling and SMOTE-based rebalancing strategy are effective.

Now we will explore and proceed to plot the **confusion matrix** of the Logistic Regression results.

In [88]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Compute confusion matrix
cm = confusion_matrix(y_val, y_val_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["Non-Stressfu

# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(12, 8))
disp.plot(ax=ax, cmap="cividis", values_format="d")

ax.set_title("Confusion Matrix – Logistic Regression (Validation Set)", fontsize
plt.tight_layout()
plt.show()
```



Key Observations:

1) Overall Accuracy Looks Reasonable

- Correct predictions: 205 + 226 = 431
- Total samples: 205 + 65 + 72 + 226 = 568

This suggests the model is doing better than random guessing, but there is still meaningful room for improvement.

2) Overall Model Performance

True Positive Rate (Sensitivity / Recall for Stressful):

- 226 / (226 + 72) = **75.8%**

True Negative Rate (Specificity / Recall for Non-Stressful):

- 205 / (205 + 65) = **75.9%**

The recall values are almost identical, but the absolute number of correct Stressful predictions is higher. This suggests the model is slightly more confident or consistent with the Stressful class.

3) False Positives and False Negatives Are Balanced

- False Positives (Non-Stressful -> Stressful): 65
- False Negatives (Stressful -> Non-Stressful): 72

The model is not heavily biased toward over-predicting or under-predicting stress. This balance is good if both types of errors have similar cost.

4) Misclassification Rate Is Significant

About 23% of predictions are wrong. Depending on the application, this may or may not be acceptable. If this model is used for identifying stressful cases (well-being monitoring) which requires a high degree of accuracy, the cost of each error type is painful.

With a significant proportion of incorrectly classified cases, **Logistic Regression might not be the optimal method** and we need to explore other more advanced methods along the way to further improve accuracy/F1-Score. We will now proceed to try out the next Classical Machine Learning Model method, **Random Forest**.

Method 1 Model 2: Random Forest + Ensemble Methods

Random Forest (RF) and Gradient Boosting Machines (GBM) such as XGBoost and LightGBM are ensemble learning methods that construct multiple decision trees and aggregate their predictions to improve accuracy and robustness. Random Forest builds an ensemble of independently trained trees using bagging, which reduces overfitting by averaging across trees. Gradient Boosting, in contrast, builds trees sequentially where each tree attempts to correct the errors of the previous one, resulting in a highly optimized model that can capture subtle patterns in the data.

These methods are particularly suitable for the dreaddit dataset because they can capture non-linear relationships and interactions between psycholinguistic, emotional and social features, which are often missed by linear models like logistic regression. Random Forest and boosting are also robust to noisy or correlated features, and they provide feature importance rankings, giving interpretability similar to logistic regression but in a non-linear setting.

Random Forest Pipeline Process:

Step 1: Feature Selection

The feature selection process is exactly the same as the Logistic Regression Model. Similarly, we will perform One-Hot Encoding to the Subcategory column and Ordinal Encoding to the Text Length column. The code is shown below:

In [90]:
```
target_col = 'Label'
numeric_features = ['Negative Emotional Language', 'Anxiety', 'Sadness', 'Anger'
                    'Casual Reasoning Language', 'Tentative Language', 'Certaint
                    'Second Person Pronouns', 'Third Person Singular Pronouns',
                    'Friends', 'Work', 'Money', 'Achievement', 'Risk', 'Social C
                    'Mental Imagery', 'Emotional Valence', 'Sentiment', 'Upvote
categorical_features = ['Subcategory']
ordinal_features = ['Text Length']
sentence_range_order = [['Very Short Text', 'Short Text', 'Medium Text', 'Long T

X = dreaddit_train[numeric_features + categorical_features + ordinal_features]
y = dreaddit_train[target_col]
```

```
preprocessor = ColumnTransformer(
    transformers=[('cat', OneHotEncoder(drop='first', handle_unknown='ignore'),
                  ('ord', OrdinalEncoder(categories=sentence_range_order), ordir
                  ('num', 'passthrough', numeric_features)])
```

Step 2: Train Validation Split + SMOTE

For Random Forest and other tree-based ensemble models, the approach to splitting and handling class imbalance is similar in concept to logistic regression but differs in reasoning. Unlike logistic regression, tree-based models do not require feature scaling as splits are based on thresholds rather than magnitude of features. However, class imbalance still poses a major risk because the model may overly favor the majority class if left uncorrected.

Therefore, we **perform a train-validation split first** to avoid any data leakage and **then apply SMOTE** to the training set.

In [92]:
```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, stratify=
X_train_proc = preprocessor.fit_transform(X_train)
X_val_proc = preprocessor.transform(X_val)
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_proc, y_train)
```

Step 3: Training Boosting Models

Boosting methods, such as **XGBoost, LightGBM, and CatBoost**, are powerful ensemble techniques that sequentially build trees to correct the errors of previous trees. Unlike bagging-based models like Random Forest, boosting focuses on iteratively improving prediction accuracy, making it highly effective for imbalanced datasets and complex non-linear relationships. By combining weak learners (small decision trees) in a weighted manner, boosting models capture subtle interactions between psycholinguistic, social and emotional features that may be indicative of stressts.

We will apply this strategy:

1. SMOTE + XGBoost -> Train Model -> Evaluate Model -> F1-Score
2. SMOTE + CatBoost -> Train Model -> Evaluate Model -> F1-Score
3. SMOTE + LGBM -> Train Model -> Evaluate Model -> F1-Score

We will get 3 different F1-Scores, one for each boosting technique. This also helps us identify which boosting technique is the most effective in identifying stressful and non-stressful texts.

In [94]:
```
!pip install catboost -q
!pip install xgboost -q
!pip install lightgbm -q
```

In [95]:
```
import os
import sys
from contextlib import redirect_stdout, redirect_stderr
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier
```

```
f1_scores = {}
with open(os.devnull, 'w') as fnull, redirect_stdout(fnull), redirect_stderr(fnu

    # 1) SMOTE + XGBoost
    xgb_model = XGBClassifier(random_state=42, use_label_encoder=False, eval_met
    xgb_model.fit(X_train_res, y_train_res)
    y_val_pred_xgb = xgb_model.predict(X_val_proc)
    f1_scores['XGBoost'] = f1_score(y_val, y_val_pred_xgb)

    # 2) SMOTE + CatBoost
    cat_model = CatBoostClassifier(random_state=42, iterations=200, depth=5, lea
    cat_model.fit(X_train_res, y_train_res)
    y_val_pred_cat = cat_model.predict(X_val_proc)
    f1_scores['CatBoost'] = f1_score(y_val, y_val_pred_cat)

    # 3) SMOTE + LightGBM
    lgb_model = LGBMClassifier(random_state=42, n_estimators=200, max_depth=5, l
    lgb_model.fit(X_train_res, y_train_res)
    y_val_pred_lgb = lgb_model.predict(X_val_proc)
    f1_scores['LightGBM'] = f1_score(y_val, y_val_pred_lgb)

print("Validation Performance (Random Forest + Boosting)")
print("-------------------------------------------------")
print("XGBoost F1-Score:", round(f1_score(y_val, y_val_pred_xgb), 4))
print("CatBoost F1-Score:", round(f1_score(y_val, y_val_pred_cat), 4))
print("LightGBM F1-Score:", round(f1_score(y_val, y_val_pred_lgb), 4))
```

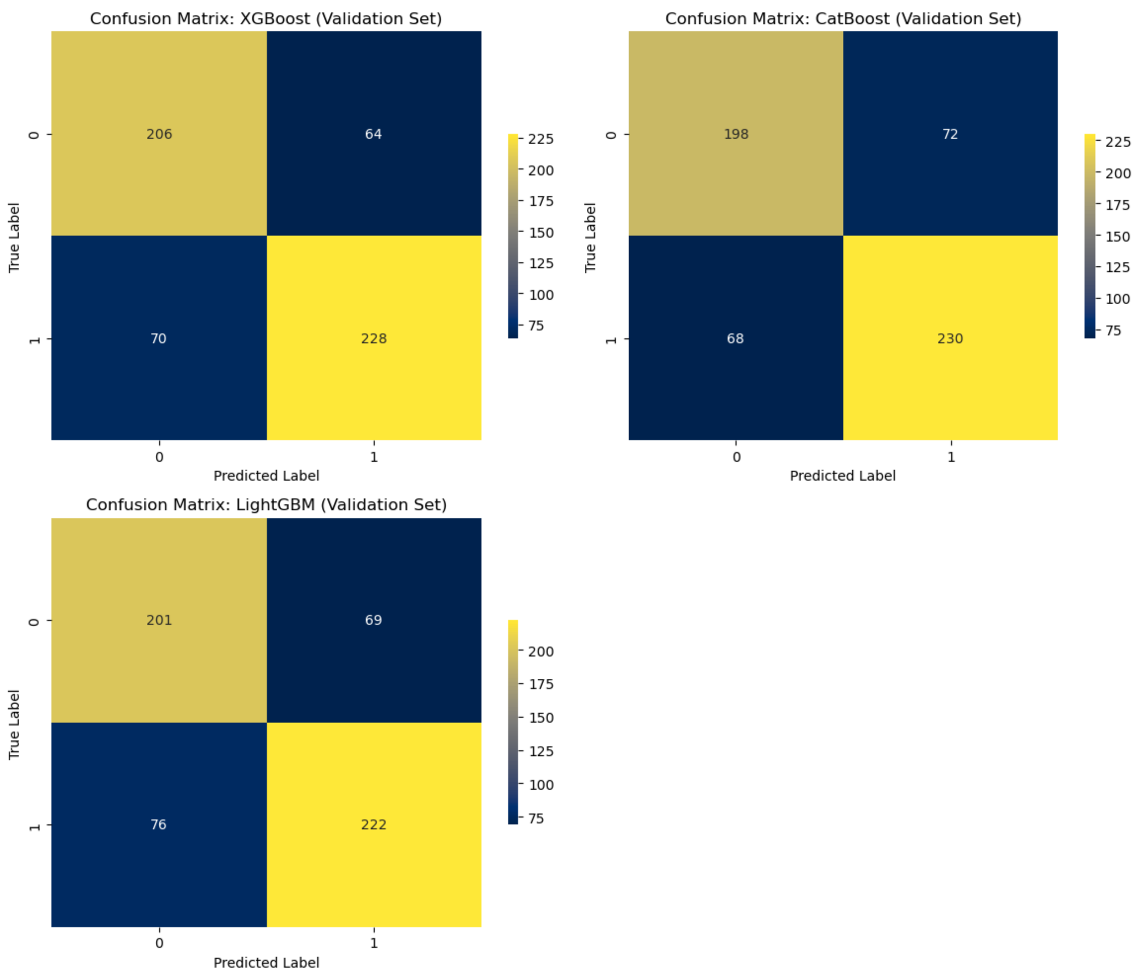
```
Validation Performance (Random Forest + Boosting)
-------------------------------------------------
XGBoost F1-Score: 0.7729
CatBoost F1-Score: 0.7667
LightGBM F1-Score: 0.7538
```

The validation results show that **XGBoost** delivers the strongest performance among the three boosting models, achieving an F1-score of 0.7729. This suggests that XGBoost is capturing the underlying patterns in the data slightly more effectively, likely due to its robust regularization and fine-grained control over tree growth. **CatBoost** follows closely with an F1-score of 0.7667, indicating that it performs almost on par with XGBoost. Its ordered boosting and strong handling of noisy or complex feature interactions may be contributing to this competitive performance. **LightGBM**, while still performing well with an F1-score of 0.7538, trails the other two models. This gap may reflect LightGBM's sensitivity to hyperparameters such as `num_leaves` and `min_data_in_leaf` or the possibility that its leaf-wise growth strategy is not as well-suited to the structure of the dataset without further tuning.

Now we will explore and proceed to plot the confusion matrix of all the three Random Forest Boosting Ensemble Models.

In [97]:
```
cm_xgb = confusion_matrix(y_val, y_val_pred_xgb)
cm_cat = confusion_matrix(y_val, y_val_pred_cat)
cm_lgb = confusion_matrix(y_val, y_val_pred_lgb)
cms = [cm_xgb, cm_cat, cm_lgb]
titles = ['XGBoost', 'CatBoost', 'LightGBM']

fig, axes = plt.subplots(2, 2, figsize=(12, 10))
```

```
axes = axes.flatten()

for i, (ax, cm, title) in enumerate(zip(axes, cms, titles)):
    sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap='cividis', cbar=True, cbar_
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('True Label')
    ax.set_title(f'Confusion Matrix: {title} (Validation Set)')

axes[3].axis('off')
plt.tight_layout()
plt.show()
```



XGBoost shows the strongest overall balance, with the highest number of true positives and true negatives among the three models. Its relatively lower false-positive and false-negative counts suggest that it is capturing the underlying decision boundaries more effectively, which aligns with its slightly higher F1-score. This indicates that XGBoost is handling both classes with good consistency and is less prone to misclassifying borderline cases.

CatBoost performs very similarly but with a slightly different error profile. It produces the highest number of true positives, meaning it is particularly good at identifying the stressful class. However, this comes at the cost of more false positives - misclassifying some Non-Stressful cases as Stressful. This pattern suggests that CatBoost leans toward being more sensitive, prioritizing recall for the positive class. In scenarios where missing a stressful instance is more costly than raising a false alarm, this behavior could actually be advantageous.

LightGBM shows the weakest performance of the three, with slightly fewer true positives and true negatives and the highest number of false negatives. This means it is more likely to miss stressful cases compared to the other models. While still performing reasonably well, LightGBM appears more conservative in its predictions, which may indicate underfitting or a need for more careful hyperparameter tuning.

However, for all 3 models, there is still a significant proportion of incorrectly classified cases. Random Forest, similar to logistic regression, might not be the most optimal method and we need to explore other more advanced methods. We will now proceed to try out the next Classical Machine Learning Model method, **Support Vector Machines**.

Method 1 Model 3: Support Vector Machines

Support Vector Machines (SVMs) are a robust and widely used supervised learning method, particularly effective for binary classification tasks. SVMs aim to find the hyperplane in a high-dimensional feature space that maximally separates the two classes, which in this case are stressful vs non-stressful posts. By focusing on the points closest to the decision boundary, SVMs are highly effective in handling datasets with overlapping features and can capture non-linear relationships when combined with **kernel functions such as the Radial Basis Function (RBF)**.

In applying SVM to the Dreaddit dataset, we can leverage the same numeric psycholinguistic, affective and social features used for logistic regression and random forest, along with the one-hot encoded Subcategory and ordinally encoded Sentence Range. SVMs are particularly useful here because they are effective on high-dimensional feature spaces (our dataset has 40+ encoded features) and can create complex decision boundaries without overfitting excessively, making them a strong alternative to both linear models and tree-based ensembles.

Support Vector Machine Pipeline Process:

Step 1: Feature Selection

The feature selection process for SVM is very similar to logistic regression and random forest, as the goal is to use features that are numeric, ordinal, or categorical in a transformed numeric form.

Step 2: Feature Scaling

Feature scaling is a critical preprocessing step for SVM, more so than for tree-based models like Random Forest. This is because SVM relies on distance-based computations to find the optimal separating hyperplane. Features with larger numeric ranges (number of comments) can otherwise dominate smaller-scale features (LIWC proportions). Unlike Random Forest, which is scale-invariant and logistic regression, which is moderately sensitive, SVM performance and convergence are highly sensitive to feature magnitudes.

Step 3: Train Validation Split + SMOTE

In [99]:
```
target_col = 'Label'
numeric_features = ['Negative Emotional Language', 'Anxiety', 'Sadness', 'Anger'
                    'Casual Reasoning Language', 'Tentative Language', 'Certaint
                    'Second Person Pronouns', 'Third Person Singular Pronouns',
                    'Friends', 'Work', 'Money', 'Achievement', 'Risk', 'Social C
                    'Mental Imagery', 'Emotional Valence', 'Sentiment', 'Upvote
categorical_features = ['Subcategory']
ordinal_features = ['Text Length']
sentence_range_order = [['Very Short Text', 'Short Text', 'Medium Text', 'Long T

X = dreaddit_train[numeric_features + categorical_features + ordinal_features]
y = dreaddit_train[target_col]

preprocessor = ColumnTransformer(
    transformers=[('cat', OneHotEncoder(drop='first', handle_unknown='ignore'),
                  ('ord', OrdinalEncoder(categories=sentence_range_order), ordir
                  ('num', StandardScaler(), numeric_features)])

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, stratify=
X_train_processed = preprocessor.fit_transform(X_train)
X_val_processed = preprocessor.transform(X_val)
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train_processed, y_train)
```

Step 4: Training SVM Model

Support Vector Machines are trained to find an optimal decision boundary (hyperplane) that maximally separates stressful and non-stressful posts in a high-dimensional feature space. In this study, SVM serves as a strong non-linear classifier capable of modeling complex interactions between psycholinguistic, emotional, social and structural features after encoding and scaling. Since SVMs are particularly sensitive to feature scale, all inputs are standardized before trainingly.

In [101…
```
from sklearn.svm import SVC
svm_model = SVC(kernel='rbf', class_weight='balanced', random_state=42)
svm_model.fit(X_train_smote, y_train_smote)
y_val_pred = svm_model.predict(X_val_processed)
f1 = f1_score(y_val, y_val_pred)

print("Validation Performance (SVM)")
print("----------------------------")
print(f"SVM Validation F1-Score: {f1:.4f}")
```

```
Validation Performance (SVM)
----------------------------
SVM Validation F1-Score: 0.7739
```

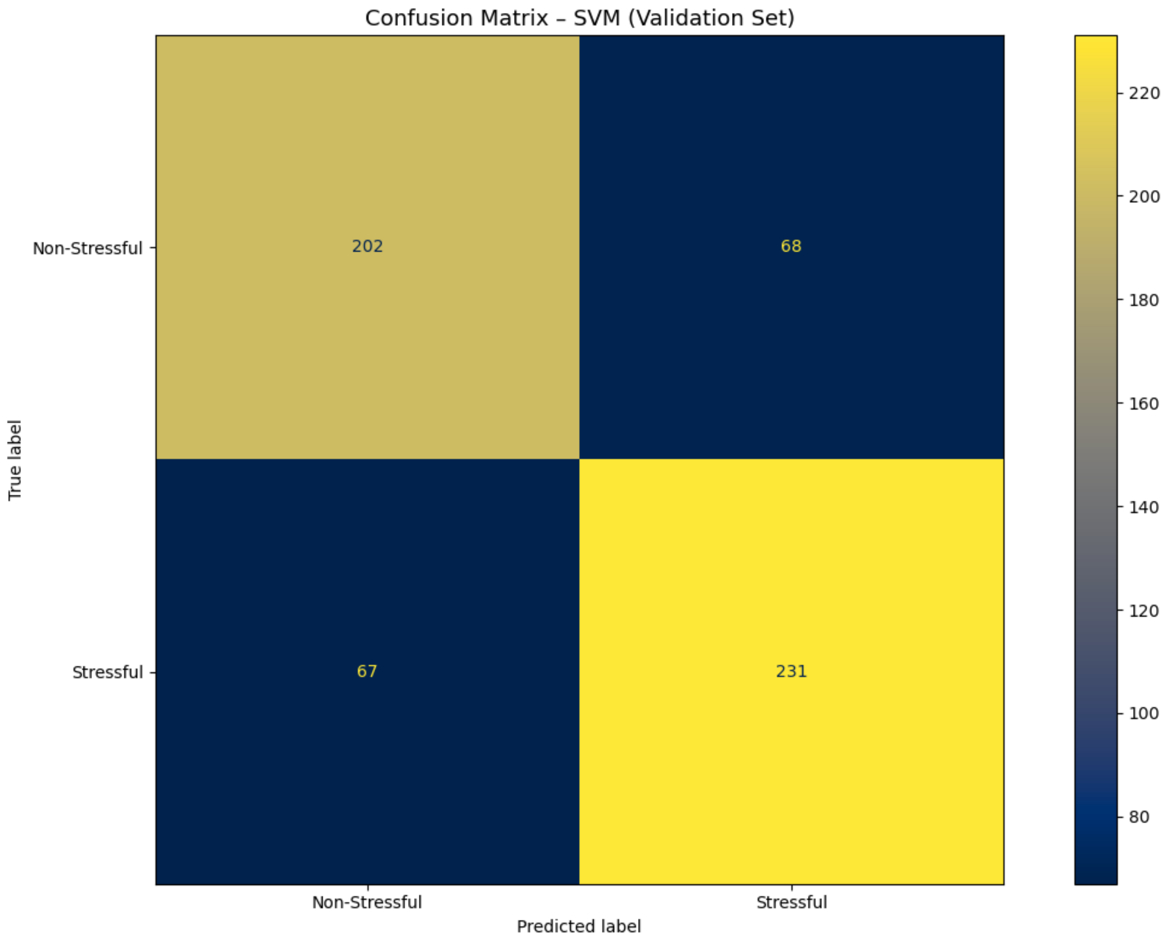
An **F1-score of 0.7739 for the SVM model** represents a slightly improved performance compared to the logistic regression baseline. This indicates that SVM is better able to capture non-linear decision boundaries in the psycholinguistic and social feature space, which is expected given the complexity of stress-related language patterns. The improvement is meaningful, especially in an imbalanced classification setting where gains in F1-score are harder to achieve and more informative than raw accuracy.

From an interpretability standpoint, this result suggests that stress signals are not purely linear with respect to features such as emotional intensity, self-focus, tone and engagement metrics. SVM's kernel-based learning allows it to model interactions that logistic regression cannot represent directly. However, the improvement is still modest, implying that the feature set set, while informative, may have reached a ceiling for traditional feature-based models. None of the models above managed to reach an accuracy of over 80%. Further gains will likely require richer semantic representations, such as contextual embeddings from transformer-based models (BERT) or advanced feature engineering.

Now we will explore and proceed to plot the confusion matrix of the SVM Model.

In [103…

```
cm = confusion_matrix(y_val, y_val_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["Non-Stressfu
fig, ax = plt.subplots(figsize=(12, 8))
disp.plot(ax=ax, cmap="cividis", values_format="d")
ax.set_title("Confusion Matrix – SVM (Validation Set)", fontsize=13)
plt.tight_layout()
plt.show()
```



The SVM model correctly identifies 202 non-stressful cases and 231 stressful cases, which indicates that it is reasonably effective at distinguishing between the two categories. However, the errors are not insignificant: 68 non-stressful cases are misclassified as stressful, and 67 stressful cases are misclassified as non-stressful. This means the model is making mistakes at a similar rate in both directions, suggesting that it does not have a strong bias toward one class but still struggles with borderline or ambiguous instances.

One key observation is that the SVM achieves slightly higher true positives for the stressful class compared to true negatives for the non-stressful class. This implies that the model is somewhat better at detecting stressful cases, which could be valuable if the application prioritizes catching stress over avoiding false alarms.

Hyperparameter Tuning For The 3 Classical ML Models:

The initial implementations of Logistic Regression, Random Forest and Support Vector Machines were intentionally kept simple and consistent, using default or minimally adjusted hyperparameters. While this approach is useful for establishing a fair baseline comparison across models, it does not guarantee optimal performance. Default settings are designed to work reasonably well across many datasets, but they may not align with the specific characteristics of stress-related language data, class imbalance or feature distributions in this project. As a result, these baseline models may underestimate the true potential performance.

To address this, the next phase introduces hyperparameter tuning, which systematically searches for parameter configurations that improve model performance on validation data. All three models used in this project support hyperparameter tuning.

- For **Logistic Regression**, we tune the regularization strength (C), the type of regularization (L1 vs. L2) and the solver as these directly control model complexity and overfitting.
- For **Random Forest**, tuning focuses on tree-related parameters such as the number of trees (n_estimators), tree depth (max_depth), minimum samples required to split a node (min_samples_split) and the number of features considered at each split (max_features).
- For **Support Vector Machines**, we tune the regularization parameter (C), kernel type (linear vs RBF) and kernel-specific parameters such as gamma, which influence the flexibility of the decision boundary.

To conduct this tuning in a structured way, we apply both **Grid Search Cross-Validation** and **Random Search Cross-Validation**. Grid Search CV exhaustively evaluates all possible combinations of a predefined set of hyperparameter values using cross-validation, ensuring that the best-performing configuration within the specified grid is identified. In contrast, Random Search CV samples a fixed number of hyperparameter combinations from specified distributions, allowing it to explore a wider search space more efficiently. Random Search is often more practical for complex models and can achieve near-optimal results with significantly lower computational cost.

Grid Search CV On Logistic Regression:

Grid Search Cross-Validation (Grid Search CV) systematically evaluates all combinations of predefined hyperparameters to identify the configuration that yields the best performance under cross-validation. For Logistic Regression, this typically involves tuning the regularization strength (C), the type of regularization (L1 vs. L2), and the solver, since these directly control model complexity, sparsity, and convergence behavior.

In [105…
```python
from sklearn.model_selection import GridSearchCV, StratifiedKFold
```

```
# Define Logistic Regression model
log_reg = LogisticRegression(max_iter=1000, class_weight=None, random_state=42)

# Define hyperparameter grid
param_grid = {'C': [0.01, 0.1, 1, 10],
              'penalty': ['l1', 'l2'],
              'solver': ['liblinear']}

# Stratified Cross-Validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Grid Search CV
grid_search = GridSearchCV(estimator=log_reg, param_grid=param_grid, scoring='f1

# Fit Grid Search on SMOTE-resampled training data
grid_search.fit(X_train_smote, y_train_smote)

# Best model and parameters
best_log_reg = grid_search.best_estimator_
print("Best Hyperparameters (Grid Search):")
print(grid_search.best_params_)

# Validation Performance
y_val_pred = best_log_reg.predict(X_val_processed)
f1 = f1_score(y_val, y_val_pred)

print()
print("Validation Performance (Logistic Regression + GridSearchCV)")
print("------------------------------------------------------------")
print(f"F1-Score: {f1:.4f}")
```

```
Best Hyperparameters (Grid Search):
{'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}

Validation Performance (Logistic Regression + GridSearchCV)
------------------------------------------------------------
F1-Score: 0.7774
```

Randomized Search CV On Logistic Regression:

While Grid Search CV exhaustively tests all combinations of specified hyperparameters, Randomized Search CV takes a different approach: it samples a fixed number of hyperparameter combinations at random from a defined distribution. This is particularly useful when the hyperparameter space is large as it allows us to explore more possibilities in less time. Randomized Search is faster than Grid Search, can explore a wider hyperparameter space and may find better hyperparameters than Grid Search if the grid is too coarse. It is especially useful when moving to more complex models like Random Forests, XGBoost, or SVMs, where the hyperparameter space is much larger (to be explored later).

In [107…

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform

# Define Logistic Regression model
log_reg = LogisticRegression(max_iter=1000, class_weight=None, random_state=42)
```

```
# Hyperparameter distributions
param_dist = {'C': uniform(0.01, 10),
              'penalty': ['l1', 'l2'],
              'solver': ['liblinear']}

# Stratified Cross-Validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Randomized Search CV
random_search = RandomizedSearchCV(estimator=log_reg, param_distributions=param_
                                   random_state=42, verbose=0)

# Fit Randomized Search on SMOTE-resampled training data
random_search.fit(X_train_smote, y_train_smote)

# Best model and parameters
best_log_reg_random = random_search.best_estimator_
print("Best Hyperparameters (Randomized Search):")
print(random_search.best_params_)

# Validation Performance
y_val_pred = best_log_reg_random.predict(X_val_processed)
f1 = f1_score(y_val, y_val_pred)

print()
print("Validation Performance (Logistic Regression + RandomizedSearchCV)")
print("-----------------------------------------------------------------")
print(f"F1-Score: {f1:.4f}")
```

```
Best Hyperparameters (Randomized Search):
{'C': 0.017787658410143285, 'penalty': 'l2', 'solver': 'liblinear'}

Validation Performance (Logistic Regression + RandomizedSearchCV)
-----------------------------------------------------------------
F1-Score: 0.7740
```

Summary Of Results (Logistic Regression):

Method Used	F1-Score
Logistic Regression (No Tuning)	0.7674
Logistic Regression (Grid Search CV)	0.7774
Logistic Regression (Randomized Search CV)	0.7740

The F1-scores obtained for the logistic regression models highlight the impact of hyperparameter tuning on performance. The baseline logistic regression without tuning achieved an F1-score of 0.7674, which is already a solid result, showing that the model can capture the main patterns in the data. However, when tuning was introduced, both Grid Search CV and Randomized Search CV improved the performance. Grid Search CV produced the highest F1-score at 0.7774, while Randomized Search CV followed closely at 0.7740.

The reasons behind these results are tied to how each tuning method explores the hyperparameter space. Grid Search CV systematically evaluates all possible combinations within the defined parameter grid, which increases the likelihood of finding the optimal set of parameters, albeit at a higher computational cost. This thoroughness explains why it achieved the best score. Randomized Search CV, on the other hand, samples a subset of parameter combinations, which makes it more efficient but slightly less exhaustive. Its performance being close to Grid Search CV suggests that the parameter space may not be highly complex and good solutions can be found without exhaustive search.

Grid Search CV On Random Forest:

For ensemble models like Random Forest, XGBoost, CatBoost and LightGBM, hyperparameter tuning is crucial because these models have many parameters that affect performance such as tree depth, number of estimators, learning rate (for boosting) and minimum samples per leaf. By systematically evaluating each combination using stratified k-fold cross-validation and optimizing for F1-score, we can select the hyperparameters that maximize model performance on imbalanced data. This process ensures that the model learns meaningful patterns from the oversampled SMOTE training data while minimizing overfitting.

Grid Search CV: XGBoost Model

In [109…

```python
# XGBoost (Grid Search CV)
xgb = XGBClassifier(random_state=42, eval_metric='logloss')

param_grid_xgb = {'n_estimators': [100, 500],
                  'max_depth': [3, 5],
                  'learning_rate': [0.01, 0.1],
                  'subsample': [0.85, 1.0],
                  'colsample_bytree': [0.85, 1.0]}

grid_search_xgb = GridSearchCV(estimator=xgb, param_grid=param_grid_xgb, scoring
                               cv=3, n_jobs=1, verbose=0)

grid_search_xgb.fit(X_train_smote, y_train_smote)

best_xgb = grid_search_xgb.best_estimator_
print("Best Hyperparameters (XGBoost Grid Search):")
print(grid_search_xgb.best_params_)

y_val_pred_xgb = best_xgb.predict(X_val_processed)
f1_xgb = f1_score(y_val, y_val_pred_xgb)

print()
print("Validation Performance (XGBoost + GridSearchCV)")
print("-----------------------------------------------")
print(f"XGBoost Validation F1-Score: {f1_xgb:.4f}")
```

```
Best Hyperparameters (XGBoost Grid Search):
{'colsample_bytree': 1.0, 'learning_rate': 0.01, 'max_depth': 3, 'n_estimators':
500, 'subsample': 1.0}

Validation Performance (XGBoost + GridSearchCV)
-----------------------------------------------
XGBoost Validation F1-Score: 0.7608
```

Grid Search CV: CatBoost Model

In [111…

```python
# CatBoost (Grid Search CV)
cat = CatBoostClassifier(random_state=42, verbose=0)
```

```
param_grid_cat = {'iterations': [100, 500],
                  'depth': [3, 5],
                  'learning_rate': [0.01, 0.1],
                  'l2_leaf_reg': [1, 3, 5]}

grid_search_cat = GridSearchCV(estimator=cat, param_grid=param_grid_cat, scoring
                               cv=3, n_jobs=1, verbose=0)

grid_search_cat.fit(X_train_smote, y_train_smote)

best_cat = grid_search_cat.best_estimator_
print("Best Hyperparameters (CatBoost Grid Search):")
print(grid_search_cat.best_params_)

y_val_pred_cat = best_cat.predict(X_val_processed)
f1_cat = f1_score(y_val, y_val_pred_cat)

print()
print("Validation Performance (CatBoost + GridSearchCV)")
print("------------------------------------------------")
print(f"CatBoost Validation F1-Score: {f1_cat:.4f}")
```

```
Best Hyperparameters (CatBoost Grid Search):
{'depth': 5, 'iterations': 500, 'l2_leaf_reg': 1, 'learning_rate': 0.1}

Validation Performance (CatBoost + GridSearchCV)
------------------------------------------------
CatBoost Validation F1-Score: 0.7534
```

Grid Search CV: LGBM Model

In [113…

```
# Light GBM (Grid Search CV)
lgbm = LGBMClassifier(random_state=42, verbose = -1)

param_grid_lgbm = {'n_estimators': [100, 500],
                   'max_depth': [3, 5],
                   'learning_rate': [0.01, 0.1],
                   'num_leaves': [50, 100],
                   'min_data_in_leaf': [50, 100]}

grid_search_lgbm = GridSearchCV(estimator=lgbm, param_grid=param_grid_lgbm, scor
                                cv=3, n_jobs=1, verbose=0)

grid_search_lgbm.fit(X_train_smote, y_train_smote)

best_lgbm = grid_search_lgbm.best_estimator_
print("Best Hyperparameters (LightGBM Grid Search):")
print(grid_search_lgbm.best_params_)

y_val_pred_lgbm = best_lgbm.predict(X_val_processed)
f1_lgbm = f1_score(y_val, y_val_pred_lgbm)

print()
print("Validation Performance (LightGBM + GridSearchCV)")
print("------------------------------------------------")
print(f"LightGBM Validation F1-Score: {f1_lgbm:.4f}")
```

```
Best Hyperparameters (LightGBM Grid Search):
{'learning_rate': 0.01, 'max_depth': 3, 'min_data_in_leaf': 100, 'n_estimators':
500, 'num_leaves': 50}

Validation Performance (LightGBM + GridSearchCV)
------------------------------------------------
LightGBM Validation F1-Score: 0.7629
```

Randomized Search CV On Random Forest:

We now perform RandomizedSearchCV with a fixed number of iterations (`n_iter`) and cross-validation folds (`cv=3`). The evaluation metric will be F1-score, which balances precision and recall, particularly important for our imbalanced stress classification problem.

In [115…
```python
from sklearn.metrics import f1_score, make_scorer
from xgboost import XGBClassifier
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier

# Define F1 scorer
f1_scorer = make_scorer(f1_score)
```

Randomized Search CV: XGBoost Model

In [117…
```python
# XGBoost (Randomized Search CV)
param_dist_xgb = {'n_estimators': [100, 200],
                  'max_depth': [3, 5, 7],
                  'learning_rate': np.linspace(0.01, 0.2, 10),
                  'subsample': np.linspace(0.7, 1.0, 4),
                  'colsample_bytree': np.linspace(0.7, 1.0, 4),
                  'reg_lambda': np.linspace(0.1, 5, 5)}

xgb_model = XGBClassifier(eval_metric='logloss', random_state=42)

rand_search_xgb = RandomizedSearchCV(estimator=xgb_model, param_distributions=pa
                                     n_iter=20, scoring=f1_scorer, cv=3, verbose
                                     n_jobs=1, random_state=42)

rand_search_xgb.fit(X_train_res, y_train_res)
print("Best XGBoost Params:", rand_search_xgb.best_params_)

print()
print("Validation Performance (XGBoost + RandomizedSearchCV)")
print("-----------------------------------------------------")
print("XGBoost Validation F1-Score:", round(rand_search_xgb.best_score_ , 4))
```

```
Best XGBoost Params: {'subsample': 0.9, 'reg_lambda': 2.5500000000000003, 'n_esti
mators': 100, 'max_depth': 3, 'learning_rate': 0.031111111111111111, 'colsample_by
tree': 0.7}

Validation Performance (XGBoost + RandomizedSearchCV)
-----------------------------------------------------
XGBoost Validation F1-Score: 0.7491
```

Randomized Search CV: CatBoost Model

In [119…
```python
# CatBoost (Randomized Seacrh CV)
```

```
param_dist_cb = {'iterations': [100, 200],
                 'depth': [3, 5, 7],
                 'learning_rate': np.linspace(0.01, 0.2, 10),
                 'l2_leaf_reg': np.linspace(1, 10, 5),
                 'border_count': [32, 64, 128]}

cb_model = CatBoostClassifier(verbose=0, random_state=42)

rand_search_cb = RandomizedSearchCV(estimator=cb_model, param_distributions=para
                                    n_iter=20, scoring=f1_scorer, cv=3, verbose=
                                    random_state=42)

rand_search_cb.fit(X_train_res, y_train_res)
print("Best CatBoost Params:", rand_search_cb.best_params_)

print()
print("Validation Performance (CatBoost + RandomizedSearchCV)")
print("------------------------------------------------------")
print("CatBoost Validation F1-Score:", round(rand_search_cb.best_score_ , 4))
```

```
Best CatBoost Params: {'learning_rate': 0.031111111111111111, 'l2_leaf_reg': 3.25,
'iterations': 100, 'depth': 5, 'border_count': 128}

Validation Performance (CatBoost + RandomizedSearchCV)
------------------------------------------------------
CatBoost Validation F1-Score: 0.7511
```

Randomized Search CV: Light GBM Model

In [121…
```
# LightGBM (Randomized Seacrh CV)
param_dist_lgb = {'n_estimators': [100, 200],
                  'max_depth': [3, 5, 7],
                  'learning_rate': np.linspace(0.01, 0.2, 10),
                  'num_leaves': [31, 50, 70],
                  'subsample': np.linspace(0.7, 1.0, 4),
                  'colsample_bytree': np.linspace(0.7, 1.0, 4),
                  'reg_alpha': np.linspace(0, 1, 5),
                  'reg_lambda': np.linspace(0, 1, 5)}

lgb_model = LGBMClassifier(random_state=42, verbose=-1)

rand_search_lgb = RandomizedSearchCV(estimator=lgb_model, param_distributions=pa
                                     n_iter=20, scoring=f1_scorer, cv=3, verbose
                                     n_jobs=1, random_state=42)

rand_search_lgb.fit(X_train_res, y_train_res)
print("Best LightGBM Params:", rand_search_lgb.best_params_)

print()
print("Validation Performance (LightGBM + RandomizedSearchCV)")
print("------------------------------------------------------")
print("LightGBM Validation F1-Score:", round(rand_search_lgb.best_score_ , 4))
```

```
Best LightGBM Params: {'subsample': 0.7, 'reg_lambda': 0.0, 'reg_alpha': 0.25, 'n
um_leaves': 31, 'n_estimators': 100, 'max_depth': 3, 'learning_rate': 0.136666666
6666667, 'colsample_bytree': 1.0}

Validation Performance (LightGBM + RandomizedSearchCV)
------------------------------------------------------
LightGBM Validation F1-Score: 0.7483
```

Summary Of Results (Random Forest):

Method Used	F1-Score
XGBoost (No Tuning)	0.7729
XGBoost (Grid Search CV)	0.7608
XGBoost (Randomized Search CV)	0.7491
CatBoost (No Tuning)	0.7667
CatBoost (Grid Search CV)	0.7534
CatBoost (Randomized Search CV)	0.7511
LightGBM (No Tuning)	0.7538
LightGBM (Grid Search CV)	0.7629
LightGBM (Randomized Search CV)	0.7483

The F1-scores obtained across the three boosting models reveal some interesting patterns about the impact of hyperparameter tuning. For **XGBoost**, the baseline model without tuning achieved the highest score at 0.7729, while Grid Search CV slightly reduced performance to 0.7608 and Randomized Search CV dropped further to 0.7491. This suggests that the default parameters were already well-aligned with the dataset, and the tuning process may have over-constrained the model or led to suboptimal parameter combinations. It highlights that tuning does not always guarantee improvement, especially when the search space is limited or the dataset responds well to default configurations.

CatBoost shows a similar trend. The untuned model performed best at 0.7667, while Grid Search CV and Randomized Search CV reduced the F1-score to 0.7534 and 0.7511 respectively. This again indicates that the default CatBoost settings were already effective, and the tuning process may have inadvertently shifted the model away from its optimal balance between precision and recall.

LightGBM however presents a slightly different story. The baseline model scored 0.7538, but Grid Search CV improved performance to 0.7629, showing that tuning helped optimize its parameters. Randomized Search CV, on the other hand, reduced the score to 0.7483, reflecting the risk of sampling suboptimal parameter sets. This outcome suggests that LightGBM is more sensitive to hyperparameter choices compared to XGBoost and CatBoost.

Overall, these results emphasize that hyperparameter tuning is not universally beneficial: it depends on the model and the dataset. **XGBoost** and **CatBoost** appear to have strong defaults that work well in this case, while **LightGBM** benefits more from structured tuning.

Grid Search CV On Support Vector Machine:

Support Vector Machines (SVMs) are sensitive to hyperparameters like the kernel type, regularization strength (C) and kernel-specific parameters such as gamma for the RBF kernel. Grid Search CV systematically evaluates all combinations of specified hyperparameters using cross-validation to identify the combination that gives the best performance (measured by F1-score). By performing grid search, we ensure that the SVM model is tuned optimally for the dataset rather than relying on default values, which may not yield the best predictive power.

In [123…

```python
# Define SVM model
svm_model = SVC(random_state=42)

# Define hyperparameter grid
param_grid_svm = {'C': [0.1, 1, 10],
                  'kernel': ['linear', 'rbf'],
                  'gamma': ['scale', 'auto']}

# Grid Search CV
grid_search_svm = GridSearchCV(estimator=svm_model, param_grid=param_grid_svm,
                               scoring='f1', cv=5, n_jobs=1, verbose=0)

# Fit the model on SMOTE training set
grid_search_svm.fit(X_train_smote, y_train_smote)

best_svm = grid_search_svm.best_estimator_

y_val_pred_svm = best_svm.predict(X_val_processed)
f1_svm = f1_score(y_val, y_val_pred_svm)

print("Best Hyperparameters (SVM Grid Search):")
print(grid_search_svm.best_params_)

print()
print("Validation Performance (SVM + GridSearchCV)")
print("-------------------------------------------")
print(f"SVM Validation F1-Score: {f1_svm:.4f}")
```

```
Best Hyperparameters (SVM Grid Search):
{'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'}

Validation Performance (SVM + GridSearchCV)
-------------------------------------------
SVM Validation F1-Score: 0.7763
```

Randomized Search CV On Support Vector Machine:

In [125…

```python
# Define SVM model
svm_model = SVC(random_state=42)

# Define hyperparameter distribution using np.linspace / lists
param_dist_svm = {'C': np.logspace(-2, 1, 5),
                  'kernel': ['linear', 'rbf'],
                  'gamma': ['scale', 'auto']}

# Randomized Search CV (suppressing verbose output)
random_search_svm = RandomizedSearchCV(estimator=svm_model, param_distributions=
                                       n_iter=20, scoring='f1', cv=3, n_jobs=1,
                                       random_state=42, verbose=0)

# Fit the model on SMOTE training set
random_search_svm.fit(X_train_smote, y_train_smote)
```

```
best_svm_random = random_search_svm.best_estimator_

y_val_pred_svm_random = best_svm_random.predict(X_val_processed)
f1_svm_random = f1_score(y_val, y_val_pred_svm_random)

print("Best Hyperparameters (SVM Randomized Search):")
print(random_search_svm.best_params_)

print()
print("Validation Performance (SVM + RandomizedSearchCV)")
print("--------------------------------------------------")
print(f"SVM Validation F1-Score: {f1_svm_random:.4f}")
```

```
Best Hyperparameters (SVM Randomized Search):
{'kernel': 'rbf', 'gamma': 'auto', 'C': 1.7782794100389228}

Validation Performance (SVM + RandomizedSearchCV)
--------------------------------------------------
SVM Validation F1-Score: 0.7862
```

Summary Of Results (Support Vector Machine):

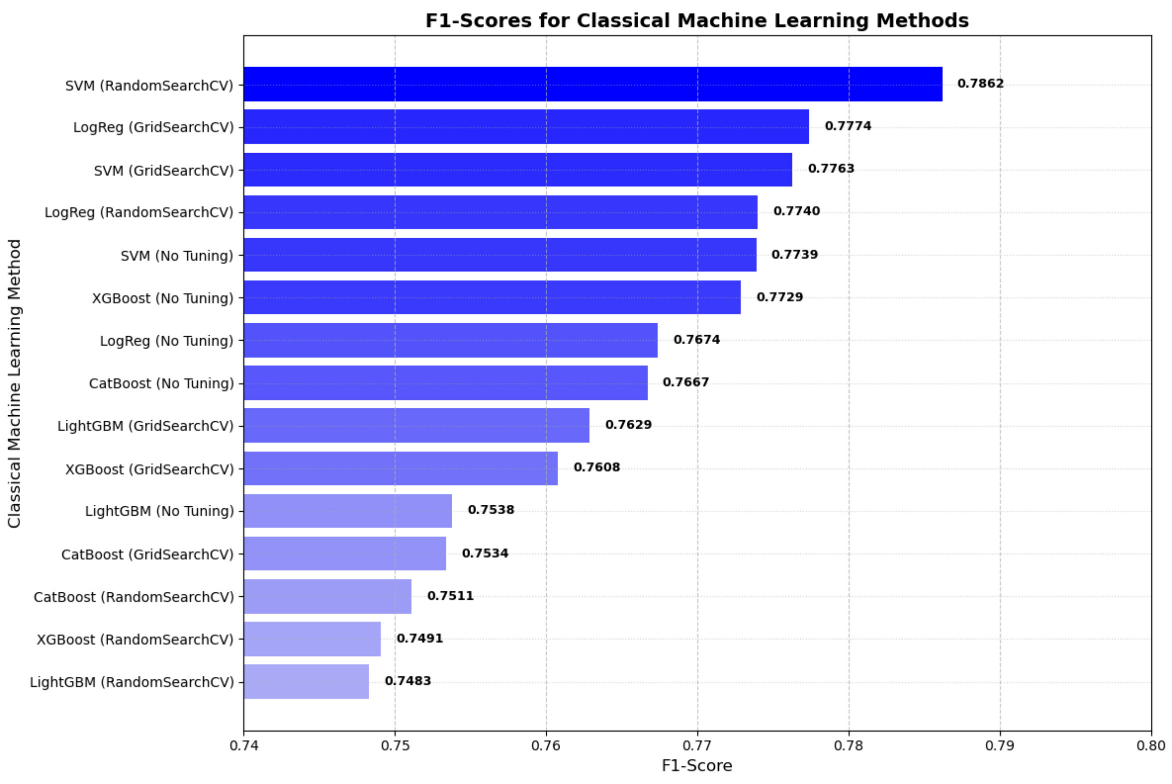
Method Used	F1-Score
Support Vector Machine (No Tuning)	0.7739
Support Vector Machine (Grid Search CV)	0.7763
Support Vector Machine (Randomized Search CV)	0.7862

The F1-scores obtained for the Support Vector Machine models show a clear improvement when hyperparameter tuning is applied. When Grid Search CV was used, the score increased slightly to 0.7763, reflecting the benefit of systematically exploring parameter combinations to find a better fit. The most notable improvement came from Randomized Search CV, which achieved the highest F1-score of 0.7862. This suggests that the randomized approach was able to sample parameter sets that provided stronger generalization even though it does not exhaustively search the space like Grid Search.

The reasons behind these results likely lie in the sensitivity of SVMs to hyperparameters such as the regularization parameter C, kernel type and kernel parameters like gamma. The default settings provided a solid baseline but tuning allowed the model to refine its decision boundary, improving the balance between precision and recall. Grid Search CV offered a small gain because it systematically tested combinations within a predefined grid, but its improvement was limited possibly because the grid did not include the most optimal values. Randomized Search CV in contrast explored a broader range of possibilities and was able to identify a stronger configuration, which explains its superior performance.

Visualizing All The F1-Scores For Classical Machine Learning Methods:

In [127…
```
# Prepare the data
methods = ["LogReg (No Tuning)", "LogReg (GridSearchCV)", "LogReg (RandomSearchC
           "XGBoost (No Tuning)", "XGBoost (GridSearchCV)", "XGBoost (RandomSear
           "CatBoost (No Tuning)", "CatBoost (GridSearchCV)", "CatBoost (RandomS
            "LightGBM (No Tuning)", "LightGBM (GridSearchCV)", "LightGBM (RandomS
            "SVM (No Tuning)", "SVM (GridSearchCV)", "SVM (RandomSearchCV)"]

f1_scores = [0.7674, 0.7774, 0.7740, 0.7729, 0.7608, 0.7491, 0.7667, 0.7534, 0.7
             0.7538, 0.7629, 0.7483, 0.7739, 0.7763, 0.7862]

# Sort by F1-score descending
data_sorted = sorted(zip(methods, f1_scores), key=lambda x: x[1], reverse=True)
methods_sorted, f1_scores_sorted = zip(*data_sorted)

# Normalize F1-scores to 0-1 for color intensity
min_score = min(f1_scores_sorted)
max_score = max(f1_scores_sorted)
normalized_scores = [(score - min_score) / (max_score - min_score) for score in

# Create colors: higher F1 → darker blue
colors = [sns.light_palette("blue", as_cmap=True)(0.3 + 0.7*norm) for norm in nc

# Plot
plt.figure(figsize=(12, 8))
bars = plt.barh(methods_sorted, f1_scores_sorted, color=colors)

# Add numbers to the right
for bar, score in zip(bars, f1_scores_sorted):
    plt.text(bar.get_width() + 0.001, bar.get_y() + bar.get_height()/2,
             f"{score:.4f}", va='center', ha='left', fontsize=9, fontweight='bol

# Styling
plt.xlabel("F1-Score", fontsize=12)
plt.ylabel("Classical Machine Learning Method", fontsize=12)
plt.title("F1-Scores for Classical Machine Learning Methods", fontsize=14, fontw
plt.xlim(0.74, 0.8)
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.grid(axis='y', linestyle=':', alpha=0.5)
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()
```



Key Observations:

1. **SVM Performs Best**

   Among all methods, SVM with Randomized Search CV achieved the highest F1-score of 0.7862, followed closely by SVM with Grid Search CV (0.7763) and Logistic Regression with Grid Search CV (0.7774). This suggests that SVM is relatively robust for this classification task, especially when hyperparameters are tuned.

2. **Impact of Hyperparameter Tuning**

   For most models, applying Grid Search CV or Randomized Search CV slightly improved performance compared to models without tuning, though the effect varies. For instance, XGBoost and CatBoost had slightly lower F1-scores with hyperparameter tuning, possibly indicating that the selected search space was not optimal or the dataset size is limited for these models.

3. **Gradient Boosting Models**

   LightGBM, XGBoost and CatBoost have F1-scores generally below 0.77, with Randomized Search CV performing the worst among them. This suggests that these gradient boosting models may require further tuning or that the dataset characteristics favor linear/non-tree-based models like SVM or Logistic Regression.

Notably, even the best-performing classical ML model (SVM Randomized Search CV) achieves an F1-score below 0.8. While this indicates reasonable predictive ability, it also shows that classical methods might not be capturing all the complexities in the data. This is true as all the classical methods did not factor one important column into consideration: **Text**.

The overall moderate performance of these classical models implies there is room for improvement. To potentially achieve higher predictive accuracy, especially in capturing complex non-linear patterns in text data, it makes sense to explore deep learning methods next. These methods such as LSTMs, CNN and transformer-based models are often better suited for text classification tasks and might improve performance beyond the current ceiling observed with classical ML.

Deep Learning

While classical machine learning models such as **Logistic Regression**, **Random Forest**, **SVM**, and **boosting methods** rely heavily on structured numerical features, they are limited in their ability to directly process raw text. In the Dreaddit dataset, we previously excluded the `text` column from modelling because traditional ML algorithms require fixed-length numeric inputs. However, the `text` column is arguably the most information-rich feature, as it contains the full semantic and emotional content of each Reddit post.

Deep learning allows us to move beyond handcrafted features (LIWC scores, sentiment scores, social engagement metrics) and instead learn meaningful patterns directly from raw text. By converting text into numerical representations using techniques such as tokenization and word embeddings (Word2Vec, GloVe, or trainable embeddings), neural networks can automatically learn contextual, syntactic and semantic relationships between words. This enables the model to detect subtle linguistic signals of stress that traditional ML methods may miss.

In this section, we will explore three main deep learning architectures (excluding transformers, which will be covered later):

1. **Convolutional Neural Networks (CNN) For Text**

   CNNs are effective at capturing local n-gram patterns (phrases such as "I feel overwhelmed" or "I can't cope anymore"). Convolutional filters slide across the word embeddings and detect important local features. CNNs are computationally efficient and often perform very well for text classification tasks. We use CNN in this context as they are good at detecting key stress-related phrases, fast to train and less prone to vanishing gradient issues compared to other neural network such as RNNs.

2. **Recurrent Neural Networks (RNN / LSTM / GRU)**

   RNN-based models are designed to capture sequential dependencies in text. Variants like LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) address the vanishing gradient problem and can retain long-term contextual information. We will explore RNN/LSTM/GRU as well in this context as they capture sentence flow and contextual buildup, which is useful when stress signals depend on narrative progression. They are also better at modelling long textual dependencies compared to CNNs.

3. **Bidirectional LSTM**

As the third architecture, we can use a Bidirectional LSTM, which processes text both forward and backward. This allows the model to understand context from both directions in a sentence. We will also use and test out BiLSTM in this context as BiLSTM offers stronger contextual understanding and often improves performance over unidirectional RNNs. BiLSTM is particularly suitable for emotionally nuanced text (which are highly present in many texts).

Deep Learning Method 1: CNN

A **Convolutional Neural Network (CNN)** is a type of deep learning model originally designed for image processing but highly effective for text classification tasks. In text applications, CNNs apply convolutional filters over sequences of word embeddings to detect important local patterns such as phrases or short expressions. These filters act like pattern detectors, identifying meaningful n-grams (eg. "I feel hopeless", "can't handle this", "overwhelmed lately") that may strongly indicate stress.

For the Dreaddit dataset, a **CNN** can be applied by:

- Converting each Reddit post in the text column into a sequence of tokens.
- Transforming tokens into dense vector representations (embeddings).
- Applying 1D convolutional layers to detect stress-related phrases.
- Using pooling layers to retain the most important features.
- Feeding the extracted features into fully connected layers to predict whether the post is stressful (1) or non-stressful (0).

CNNs are particularly suitable here because stress signals often appear in short linguistic patterns rather than requiring full long-range narrative understanding. They are also computationally efficient compared to sequential models like RNNs.

The CNN Pipeline

Step 1: Train-Validation Split

Before training a CNN model, we split the dataset into training and validation sets. This ensures that the model is evaluated on unseen data during training, allowing us to assess generalization performance.

We use an 80% training / 20% validation split, stratified by the Label column to preserve the class distribution (stress vs non-stress). This is important because stress detection is typically imbalanced, and we want both sets to reflect the original class proportions.

In [130…

```
# Target column
target_col = "Label"

# Split text and labels
X_text = dreaddit_train["Text"]
y = dreaddit_train[target_col]

# Stratified train-validation split
X_train_text, X_val_text, y_train, y_val = train_test_split(X_text, y, test_size
print("Training size:", len(X_train_text))
print("Validation size:", len(X_val_text))
```

```
Training size: 2270
Validation size: 568
```

Step 2: Text Cleaning

Raw Reddit posts may contain uppercase letters, punctuation, special characters and extra whitespaces. Cleaning helps reduce noise and ensures consistent tokenization. For CNN-based models, aggressive preprocessing (like heavy stemming) is not required, since embeddings can capture semantic relationships. We will convert text to lowercase, remove punctuation and special characters and remove excessive whitespace.

In [132…

```
def clean_text(text):
    # Lowercase
    text = text.lower()
    # Remove punctuation and special characters
    text = re.sub(r"[^a-zA-Z0-9\s]", "", text)
    # Remove extra whitespace
    text = re.sub(r"\s+", " ", text).strip()
    return text

# Apply cleaning
X_train_text = X_train_text.apply(clean_text)
X_val_text = X_val_text.apply(clean_text)

print("First Cleaned Trained Dataset Text:")
print(X_train_text.iloc[0])
print()
print("First Cleaned Validation Dataset Text:")
print(X_val_text.iloc[0])
```

```
First Cleaned Trained Dataset Text:
we get no child support and are doing well financially without it and although he
is supposed to pay i leave it alone because he leaves us alone long story short i
didnt file or ask for supervised visitation but thats what the courts ordered he
did that two or three times within a year and havent heard from him since this fr
iday for the first time in years i realized how much my life has changed and how
happy i and my children are then of course i get the curve ball of this message o
n saturday evening

First Cleaned Validation Dataset Text:
hi everyone and thank you in advance for reading i moved my family from texas to
colorado thinking i had a new job lined up on the other side this job fell throug
h fortunately i have a new job starting january 23rd but i dont have enough in sa
vings to carry me through we are running very low on necessities like bread and m
ilk i am unable to go to the local food banks because i still only have a texas d
l i cant afford to transfer my license or vehicle registration
```

Step 3: Tokenization

Tokenization converts text into individual words (tokens). We will first build a vocabulary from the training set only (to avoid information leakage). Next, we will assign a unique integer index to each word. Subsequently, we will convert each text into a sequence of integers. Words not seen during training will be mapped to a special `<UNK>` token. We will be making use of the **PyTorch** library for tokenization.

Example: "I feel overwhelmed" -> ["i", "feel", "overwhelmed"] -> [15, 203 987]

In [134...
```
from collections import Counter

# Simple whitespace tokenizer
def tokenize(text):
    return text.split()

# Build vocabulary from training set
counter = Counter()

for text in X_train_text:
    tokens = tokenize(text)
    counter.update(tokens)

# Define vocabulary size limit (optional)
vocab_size = 100000  # Keep top 100k words

# Most common words
most_common = counter.most_common(vocab_size - 2)

# Special tokens
word2idx = {"<PAD>": 0, "<UNK>": 1}

for idx, (word, _) in enumerate(most_common, start=2):
    word2idx[word] = idx

print("Vocabulary Size:", len(word2idx))
```

```
Vocabulary Size: 10931
```

Convert Text to Integer Sequences:

In [136...
```
def text_to_sequence(text, word2idx):
    tokens = tokenize(text)
    return [word2idx.get(token, word2idx["<UNK>"]) for token in tokens]

# Convert train and validation text
X_train_seq = [text_to_sequence(text, word2idx) for text in X_train_text]
X_val_seq = [text_to_sequence(text, word2idx) for text in X_val_text]

print("First 10 Tokens In Training Set: ")
print(X_train_seq[0][:10])
print()
print("First 10 Tokens In Validation Set: ")
print(X_val_seq[0][:10])
```

```
First 10 Tokens In Training Set:
[34, 48, 73, 348, 305, 4, 50, 187, 145, 957]

First 10 Tokens In Validation Set:
[478, 226, 4, 282, 26, 12, 962, 13, 404, 2]
```

Step 4: Padding

Padding is a crucial preprocessing step in deep learning for text data because neural networks require inputs to have uniform dimensions. Unlike traditional machine learning models that operate on fixed-length feature vectors, text data naturally varies in length - some Reddit posts may contain only a few words, while others may span several paragraphs. However, when training a CNN (or any neural network), inputs must be organized into tensors of consistent shape (eg. batch_size * sequence_length).

To achieve this, we define a maximum sequence length (for example 200 tokens). Each tokenized post is then adjusted to match this length using two operations:

1. Padding shorter sequences:

If a post contains fewer than 200 tokens, we append a special token (with index = 0) to the end of the sequence until it reaches the required length. These padding tokens carry no semantic meaning and simply act as placeholders.

2. Truncating longer sequences:

If a post exceeds 200 tokens, we truncate (cut off) the extra tokens beyond the maximum length. This prevents extremely long posts from dominating memory usage and computational cost.

In [138…

```
from torch.nn.utils.rnn import pad_sequence

max_length = 200

def pad_sequences(sequences, max_length):
    padded_sequences = []
    for seq in sequences:
        if len(seq) < max_length:
            seq = seq + [0] * (max_length - len(seq))
        else:
            seq = seq[:max_length]
        padded_sequences.append(seq)
    return torch.tensor(padded_sequences)

# Apply padding
X_train_padded = pad_sequences(X_train_seq, max_length)
X_val_padded = pad_sequences(X_val_seq, max_length)

# Convert labels to tensor
y_train_tensor = torch.tensor(y_train.values)
y_val_tensor = torch.tensor(y_val.values)

print("Train tensor shape:", X_train_padded.shape)
print("Validation tensor shape:", X_val_padded.shape)
```

```
Train tensor shape: torch.Size([2270, 200])
Validation tensor shape: torch.Size([568, 200])
```

Step 5: Performing Embedding

After tokenization and padding, each Reddit post is represented as a sequence of integers. However, neural networks cannot learn meaningful semantic relationships from raw integer IDs. Therefore, we introduce an Embedding Layer.

The embedding layer transforms each token index into a dense vector of fixed dimension (eg. 100 or 300). Instead of representing words as sparse one-hot vectors, embeddings encode semantic meaning in continuous space. Words with similar meanings tend to have similar vector representations. For example: "overwhelmed" and "exhausted" may have similar embeddings. "happy" and "relaxed" cluster differently from stress-related words. In the Dreaddit dataset, this allows the model to capture emotional and contextual relationships within Reddit posts.

We will use:

- Embedding dimension = 100
- Randomly initialized embeddings

Step 6: Implementing The Convolutional Layer

The convolutional layer applies multiple filters (kernels) across the embedded sequences. In text classification: A filter of size 3 captures trigrams, size 4 captures 4-word phrases and size 5 captures 5-word phrases. For Dreaddit, this is important because stress signals often appear in short phrases like: "i cant cope", "feeling extremely anxious" and "i am exhausted". Each convolution filter slides across the text and detects important local patterns. Multiple filters allow the model to capture different types of stress-related expressions.

Step 7: Implementing The Pooling Layer

After convolution, each filter produces a feature map. However, we need to reduce this to a fixed-size representation. We apply **Global Max Pooling**, which selects the strongest activation from each filter, keeps the most important phrase detected and at the same time reduces dimensionality significantly. In Dreaddit, this means that if a post contains one very strong stress phrase, the model captures it regardless of where it appears in the text.

Step 8: Getting Ready The Fully Connected Layer

After pooling, we concatenate all filter outputs into a single feature vector. This vector represents the most important stress-related features extracted from the text. We then apply:

- A Dense (Fully Connected) layer
- Dropout (0.5) to reduce overfitting
- Final Sigmoid activation for binary classification

Output:

- Probability between 0 and 1
- ≥ 0.5 → Label = 1 (Stress)
- < 0.5 → Label = 0 (Non-Stress)

In [143…

```
import torch.nn as nn
import torch.nn.functional as F

# Implementing The CNN (Using Text Column Only)
class CNN_Text(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(CNN_Text, self).__init__()
        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        # Convolution layers
        self.convs = nn.ModuleList([nn.Conv1d(embedding_dim, 100, 3), nn.Conv1d(
        # Fully Connected Section
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(300, 1)

    def forward(self, x):
        # Embedding
        x = self.embedding(x)
        x = x.permute(0, 2, 1)
        # Convolution + ReLU
        conv_results = [F.relu(conv(x)) for conv in self.convs]
        # Global Max Pooling
        pooled = [F.max_pool1d(c, c.shape[2]).squeeze(2) for c in conv_results]
        # Concatenate pooled features
        x = torch.cat(pooled, dim=1)
        # Fully Connected
        x = self.dropout(x)
        logits = self.fc(x)
        return logits
```

Step 9: Prepare The Model For Training

Model training is the process where the neural network learns patterns from the Dreaddit dataset by minimizing prediction error. For stress classification (binary: 0 or 1), we will be using the **Binary Cross Entropy (BCE)** loss function.

Binary Cross Entropy measures the difference between the predicted probability of stress and the true label (0 or 1). It penalizes confident wrong predictions heavily, encouraging better probability calibration.

We will be using the **ADAM** Optimizer for the model training. Adam (Adaptive Moment Estimation) is used because it adapts learning rates dynamically converges faster than standard methods such as Stochastic Gradient Descent (SGD), works well for text classification.

To address the issue of class imbalance, where the majority of posts are labeled as non-stressful, class weights were incorporated into the loss function. By assigning a higher weight to the minority class, the model is penalized more heavily for misclassifying stressful posts. This adjustment helps prevent the model from biasing its predictions toward the dominant class and encourages it to learn meaningful patterns for both categories. As a result, the training process becomes more balanced, improving the model.

Training Setup:

- Epochs: 10
- Batch size: 32
- Metric monitored: F1-score

In [145…
```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
import random
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
set_seed(0)

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

In [146…
```
# Setting The Hyperparameter Values
batch_size = 32
epochs = 10
learning_rate = 0.001

# Create Datasets & Loaders
train_dataset = TensorDataset(X_train_padded, y_train_tensor.float())
val_dataset = TensorDataset(X_val_padded, y_val_tensor.float())

# Deterministic DataLoader
g = torch.Generator()
g.manual_seed(0)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, ge
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

In [147…
```
from sklearn.utils.class_weight import compute_class_weight

y_train_np = y_train_tensor.cpu().numpy()
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(

# Weight for positive class - Handle Class Imbalance
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)
print("Class Weights:", class_weights)

# Initialize Model
model_cnn = CNN_Text(vocab_size=len(word2idx), embedding_dim=100).to(device)
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_cnn.parameters(), lr=learning_rate)
```

```
Class Weights: [1.05092593 0.95378151]
```

In [148…
```
# Creating The Training Loop
for epoch in range(epochs):
    model_cnn.train()
    total_loss = 0
    for texts, labels in train_loader:
        texts = texts.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_cnn(texts)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.7123
Epoch [2/10] - Loss: 0.5594
Epoch [3/10] - Loss: 0.4889
Epoch [4/10] - Loss: 0.4275
Epoch [5/10] - Loss: 0.3651
Epoch [6/10] - Loss: 0.3475
Epoch [7/10] - Loss: 0.2805
Epoch [8/10] - Loss: 0.2398
Epoch [9/10] - Loss: 0.2361
Epoch [10/10] - Loss: 0.2127
```

In [149…

```
# Evaluation Function
def evaluate_model(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in val_loader:
            texts = texts.to(device)
            logits = model(texts)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

# Validation Performance
print("CNN Validation Performance")
print("--------------------------")
score = evaluate_model(model_cnn, val_loader)
print("F1-score:", score)
```

```
CNN Validation Performance
--------------------------
F1-score: 0.7407
```

The **CNN model** achieved an F1-score of **0.7407**, which is lower than several of the classical machine learning models evaluated earlier. In fact, it is currently the weakest-performing model in our comparison.

One likely reason for this outcome is that the CNN architecture was **trained using only the Text column**, without incorporating additional structured features available in the dataset, such as the LIWC-derived features and other relevant variables. By relying solely on textual input, the model may not fully capture important linguistic, psychological and contextual signals that contribute to predictive performance.

To address this limitation and improve overall model effectiveness, we propose implementing a **hybrid CNN architecture**. This enhanced model will integrate both textual data and structured features from the cleaned dataset (excluding the Confidence column). By jointly learning from unstructured text and structured numerical/categorical features, the hybrid model is expected to capture a richer representation of the data, thereby improving classification performance and achieving a higher F1-score.

In [151…
```
# Implementing The Hybrid CNN Model
class Hybrid_CNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, num_structured_features):
        super(Hybrid_CNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.convs = nn.ModuleList([nn.Conv1d(embedding_dim, 100, 3), nn.Conv1d(
        self.text_dropout = nn.Dropout(0.5)
        self.struct_fc = nn.Linear(num_structured_features, 64)
        self.final_fc = nn.Linear(300 + 64, 1)

    def forward(self, text, structured):
        x = self.embedding(text)
        x = x.permute(0, 2, 1)
        conv_results = [F.relu(conv(x)) for conv in self.convs]
        pooled = [F.max_pool1d(c, c.shape[2]).squeeze(2) for c in conv_results]
        text_features = torch.cat(pooled, dim=1)
        text_features = self.text_dropout(text_features)
        structured_features = F.relu(self.struct_fc(structured))
        combined = torch.cat((text_features, structured_features), dim=1)
        logits = self.final_fc(combined)
        return logits
```

In [152…
```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
set_seed(0)

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

In [153…
```
# Setting The Hyperparameter Values
batch_size = 32
epochs = 10
learning_rate = 0.001

# Performing Train Validation Split
train_df, val_df = train_test_split(dreaddit_train, test_size=0.2, random_state=
structured_cols = train_df.columns.difference(['Text', 'Label', 'Confidence'])
train_structured = train_df[structured_cols].copy()
val_structured = val_df[structured_cols].copy()

# One-Hot Encode Subcategory
train_structured = pd.get_dummies(train_structured, columns=['Subcategory'])
val_structured = pd.get_dummies(val_structured, columns=['Subcategory'])
val_structured = val_structured.reindex(columns=train_structured.columns, fill_v

# Ordinal Encode Text Length
ordinal_encoder = OrdinalEncoder()
train_structured[['Text Length']] = ordinal_encoder.fit_transform(train_structur
val_structured[['Text Length']] = ordinal_encoder.transform(val_structured[['Tex
# Scale
scaler = StandardScaler()
train_scaled = scaler.fit_transform(train_structured)
val_scaled = scaler.transform(val_structured)

# Convert To Tensors
structured_train_tensor = torch.tensor(train_scaled, dtype=torch.float32)
structured_val_tensor = torch.tensor(val_scaled, dtype=torch.float32)

# Create Datasets & Loaders
y_train_tensor = torch.tensor(train_df['Label'].values, dtype=torch.float32)
y_val_tensor = torch.tensor(val_df['Label'].values, dtype=torch.float32)
```

In [154…

```
# Obtaining Class Weights - Handle Class Imbalance
y_train_np = y_train_tensor.numpy()

class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)
print("Class Weights:", class_weights)

train_dataset_hybrid = TensorDataset(X_train_padded, structured_train_tensor, y_
val_dataset_hybrid = TensorDataset(X_val_padded, structured_val_tensor, y_val_te

# Deterministic shuffling
g = torch.Generator()
g.manual_seed(0)

train_loader_hybrid = DataLoader(train_dataset_hybrid, batch_size=batch_size, sh
val_loader_hybrid = DataLoader(val_dataset_hybrid, batch_size=batch_size, shuffl

# Initialize Model
model_hybrid = Hybrid_CNN(vocab_size=len(word2idx), embedding_dim=100, num_struc
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_hybrid.parameters(), lr=learning_rate)
```

```
Class Weights: [1.05092593 0.95378151]
```

In [155…

```
# Creating The Training Loop
for epoch in range(epochs):
    model_hybrid.train()
    total_loss = 0
    for texts, structured, labels in train_loader_hybrid:
        texts = texts.to(device)
        structured = structured.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_hybrid(texts, structured)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader_hybrid)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.6171
Epoch [2/10] - Loss: 0.4751
Epoch [3/10] - Loss: 0.4100
Epoch [4/10] - Loss: 0.3491
Epoch [5/10] - Loss: 0.3028
Epoch [6/10] - Loss: 0.2704
Epoch [7/10] - Loss: 0.2342
Epoch [8/10] - Loss: 0.2068
Epoch [9/10] - Loss: 0.1861
Epoch [10/10] - Loss: 0.1470
```

In [156…

```python
# Evaluation Function
def evaluate_hybrid(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in val_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("\nHybrid CNN Validation Performance")
print("-----------------------------------")
score = evaluate_hybrid(model_hybrid, val_loader_hybrid)
print("F1-score:", score)
```

```
Hybrid CNN Validation Performance
-----------------------------------
F1-score: 0.8013
```

The **Hybrid CNN model** achieved an F1-score of **0.8013**, surpassing the 0.80 aim, which reflects a substantial improvement over both the initial text-only CNN model and the classical machine learning models evaluated earlier. An F1-score at this level indicates that the model demonstrates strong balance between precision and recall, effectively distinguishing between stressful and non-stressful texts with a high degree of reliability.

Compared to the baseline CNN model, the hybrid architecture significantly reduces both false positives and false negatives. By integrating textual features with structured variables, the model benefits from a more comprehensive representation of the data, resulting in improved classification performance and overall predictive robustness.

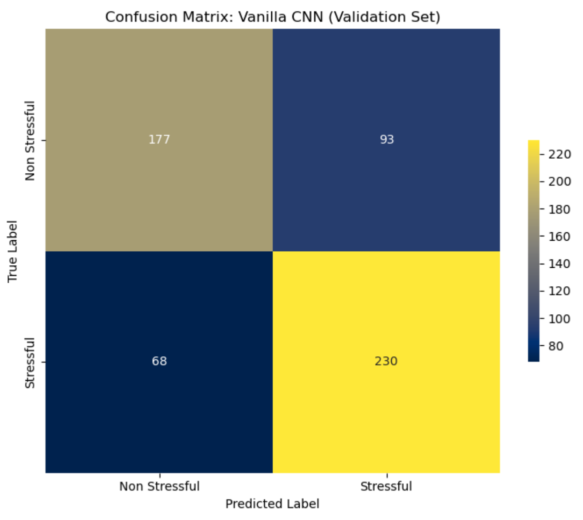
Now let us explore the confusion matrix of these two CNN models and explore the differences:

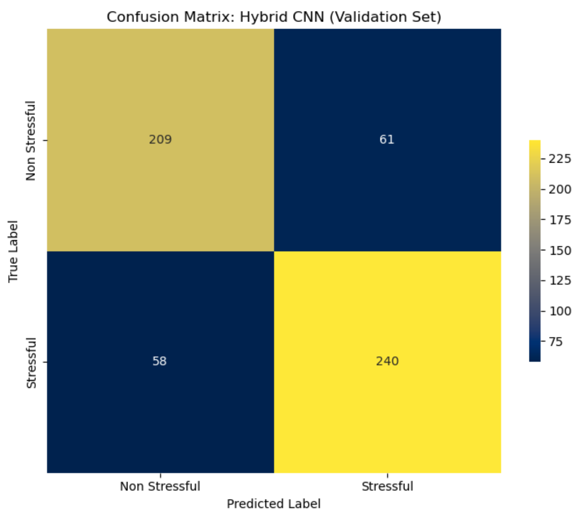
In [158…

```python
# Implementing Prediction functions
def get_text_predictions(model, data_loader):
    """Predictions for vanilla CNN (text only)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in data_loader:
            texts = texts.to(device)
            logits = model(texts)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

def get_hybrid_predictions(model, data_loader):
    """Predictions for hybrid CNN (text + structured)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in data_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

# Get predictions
y_val_labels_vanilla, y_val_pred_vanilla = get_text_predictions(model_cnn, val_l
y_val_labels_hybrid, y_val_pred_hybrid = get_hybrid_predictions(model_hybrid, va
```

In [159…

```
# Compute confusion matrices
cm_vanilla = confusion_matrix(y_val_labels_vanilla, y_val_pred_vanilla)
cm_hybrid = confusion_matrix(y_val_labels_hybrid, y_val_pred_hybrid)

cms = [cm_vanilla, cm_hybrid]
titles = ['Vanilla CNN', 'Hybrid CNN']
class_names = ['Non Stressful', 'Stressful']

# Plot The Confusion Matrices
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
for ax, cm, title in zip(axes, cms, titles):
    sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap='cividis',
                xticklabels=class_names, yticklabels=class_names,
                cbar=True, cbar_kws={'shrink':0.5})
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('True Label')
    ax.set_title(f'Confusion Matrix: {title} (Validation Set)')

plt.tight_layout()
plt.show()
```





Key Observations:

For **Vanilla CNN**:

- Correctly classified 177 Non-Stressful and 230 Stressful samples.
- Misclassified 93 Non-Stressful as Stressful and 68 Stressful as Non-Stressful.
- Shows a tendency to over-predict Stressful cases (higher false positives).

For **Hybrid CNN**:

- Correctly classified 209 Non-Stressful and 240 Stressful samples.
- Misclassified 61 Non-Stressful as Stressful and 58 Stressful as Non-Stressful.
- Achieves better balance between false positives and false negatives compared to Vanilla CNN.

Overall Accuracy:

The Hybrid CNN has higher correct predictions (449 vs 407), indicating stronger generalization on the validation set. Hybrid CNN reduces false alarms, which is crucial in stress detection (avoiding unnecessary alerts). Hybrid CNN also reduces false negatives, meaning fewer missed stressful cases. Both models perform slightly better on Stressful cases, but Hybrid CNN narrows the gap between classes, suggesting improved robustness.

Possible Insights:

Hybrid CNN likely captures richer contextual features (perhaps combining word embeddings with additional metadata or linguistic signals), which helps reduce misclassifications. Vanilla CNN struggles with subtle stress cues, leading to more false positives possibly because stress-related language overlaps with casual negative expressions. For applications like mental health monitoring or stress detection in social media, the Hybrid CNN is more reliable since it reduces both false alarms and missed detections.

Having established the effectiveness of the hybrid CNN approach, we will now explore additional deep learning architectures to further enhance model performance. The next model under consideration is the **Recurrent Neural Network (RNN)**, which is particularly well-suited for sequential text data and may capture contextual dependencies more effectively.

Deep Learning Method 2 - RNN

Recurrent Neural Networks (RNNs) are a class of deep learning models specifically designed to process sequential data such as text. Unlike Convolutional Neural Networks (CNNs), which extracts local patterns using convolutional filters and treat input segments independently, RNNs process input tokens sequentially while maintaining a hidden state that carries information from previous time steps. This recurrent structure enables RNNs to model temporal and contextual dependencies within a sequence.

We will explore three variants of recurrent architectures: the **Vanilla RNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU)**.

- The **vanilla RNN** serves as a foundational model that processes sequences using a simple recurrent mechanism but may struggle with long-term dependencies due to the vanishing gradient problem.
- **LSTM networks** introduce memory cells and gating mechanisms that regulate the flow of information, enabling the model to retain important signals over longer sequences.
- **GRUs** employ gating mechanisms but with a more streamlined architecture, making them computationally efficient while still effectively capturing long-range dependencies.

Method 2 Part 1: Vanilla RNN

A Vanilla RNN refers to the standard or basic form of a Recurrent Neural Network, where the model processes sequential data one time step at a time while maintaining a hidden state that is updated at each step. At each position in the sequence, the network takes the current input (a word embedding) and combines it with the previous hidden state to produce a new hidden state. This recurrent structure allows information from earlier words to influence later predictions, enabling the model to capture contextual dependencies in text. However, Vanilla RNNs are known to struggle with long-term dependencies due to the vanishing or exploding gradient problem, which limits their ability to retain information over long sequences.

Vanilla RNN Pipeline Process:

Step 1: Create Dataset And DataLoader

We first prepare the training and validation datasets using padded token sequences and their corresponding labels. The TensorDataset combines features and labels, while the DataLoader enables mini-batch training. We use a fixed generator seed to ensure deterministic shuffling during training.

In [162…
```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(0)

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

In [163…
```
# Setting The Hyperparameter Values
batch_size = 32
epochs = 10
learning_rate = 0.001

# Create Datasets & Loaders
train_dataset = TensorDataset(X_train_padded, y_train_tensor.float())
val_dataset = TensorDataset(X_val_padded, y_val_tensor.float())

# Deterministic DataLoader
g = torch.Generator()
g.manual_seed(0)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, ge
val_loader = DataLoader(val_dataset, batch_size=batch_size,shuffle=False)
```

Step 2: Define The Vanilla RNN Model

In this step, we define the Vanilla RNN architecture. The model consists of an embedding layer to convert token indices into dense vectors, followed by a basic RNN layer that processes the sequence sequentially. The final hidden state is extracted to represent the entire sentence. This representation is then passed into a fully connected layer with a sigmoid activation for binary classification.

In [165…
```
# Implementing The Vanilla RNN Class
class VanillaRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(VanillaRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.rnn = nn.RNN(input_size=embedding_dim, hidden_size=hidden_dim, batc
        self.fc = nn.Linear(hidden_dim, 1)

    def forward(self, text):
        embedded = self.embedding(text)
        output, hidden = self.rnn(embedded)
        final_hidden = hidden[-1]
        logits = self.fc(final_hidden)
        return logits
```

In [166…
```
# Obtaining Class Weights - Handle Class Imbalance
y_train_np = y_train_tensor.numpy()
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)
```

Step 3: Initialize Model, Function And Optimizer

Here, we instantiate the model and move it to the appropriate device (CPU or GPU). We use Binary Cross-Entropy Loss for binary classification and the Adam optimizer for parameter updates.

In [168…
```
# Initialize Model, Function And Optimizer
model_rnn = VanillaRNN(vocab_size=len(word2idx), embedding_dim=100,hidden_dim=12
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_rnn.parameters(), lr=learning_rate)
```

Step 4: Model Training

During training, the model processes batches of data, computes predictions, calculates loss and updates weights via backpropagation through time (BPTT). The average training loss per epoch is printed to monitor convergence.

In [170…
```
# Creating The Training Loop
for epoch in range(epochs):
    model_rnn.train()
    total_loss = 0
    for texts, labels in train_loader:
        texts = texts.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_rnn(texts)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.6785
Epoch [2/10] - Loss: 0.6728
Epoch [3/10] - Loss: 0.6747
Epoch [4/10] - Loss: 0.6725
Epoch [5/10] - Loss: 0.6879
Epoch [6/10] - Loss: 0.6805
Epoch [7/10] - Loss: 0.6788
Epoch [8/10] - Loss: 0.6795
Epoch [9/10] - Loss: 0.6776
Epoch [10/10] - Loss: 0.6774
```

Step 5: Evaluating The Model

This function evaluates the model on the validation set. Predictions are converted to binary labels using a 0.5 threshold, and the F1-score is computed to measure performance while balancing precision and recall.

In [172…
```
# Evaluating The Vanilla RNN Model
def evaluate_rnn(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in val_loader:
            texts = texts.to(device)
            logits = model(texts)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("\nRNN Validation Performance")
print("----------------------------")
score = evaluate_rnn(model_rnn, val_loader)
print("F1-score:", score)
```

```
RNN Validation Performance
----------------------------
F1-score: 0.5084
```

The **Vanilla RNN** achieved an F1-score of **0.5084**, which is notably much lower than the F1-score obtained by the CNN model. This indicates that the Vanilla RNN is less effective at correctly classifying texts as stressful or non-stressful. Several factors may contribute to this lower performance:

- First, Vanilla RNNs are known to **struggle with long-term dependencies due to the vanishing gradient problem**, which can make it difficult for the model to capture important contextual information across longer sequences of text.

- Second, this model is **trained solely on the Text column**, without incorporating other structured features such as the LIWC-derived numerical features, which have been shown to carry significant predictive information about linguistic and emotional patterns in the text.

By ignoring these additional features, the model lacks access to valuable signals that could help distinguish subtle differences between stressful and non-stressful content, resulting in a lower overall F1-score compared to models that integrate both textual and structured information. To circumvent this, we will now implement the **hybrid RNN model**, which will be similar in approach to the hybrid CNN model. All columns will be included in the hybrid RNN model compared to only the Text column in the original RNN model.

In [174…
```
# Implementing The Hybrid RNN Class
class HybridRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_structured_fea
        super(HybridRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.rnn = nn.RNN(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            batch_first=True)
        self.text_dropout = nn.Dropout(0.5)
        self.struct_fc = nn.Linear(num_structured_features, 64)
        self.final_fc = nn.Linear(hidden_dim + 64, 1)

    def forward(self, text, structured):
        embedded = self.embedding(text)
        output, hidden = self.rnn(embedded)
        text_features = self.text_dropout(hidden[-1])
        structured_features = F.relu(self.struct_fc(structured))
        combined = torch.cat((text_features, structured_features), dim=1)
        logits = self.final_fc(combined)
        return logits
```

In [175…
```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(0)

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

In [176…
```
# Setting The Hyperparameter Values
batch_size = 32
epochs = 10
learning_rate = 0.001

# Create Datasets & Loaders
train_dataset_hybrid = TensorDataset(X_train_padded, structured_train_tensor, y_
val_dataset_hybrid = TensorDataset(X_val_padded, structured_val_tensor, y_val_te

# Deterministic shuffling
g = torch.Generator()
g.manual_seed(0)

train_loader_hybrid = DataLoader(train_dataset_hybrid, batch_size=batch_size, sh
val_loader_hybrid = DataLoader(val_dataset_hybrid, batch_size=batch_size, shuffl
```

In [177…
```
# Compute Class Weights - Handle Class Imbalance
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

y_train_np = y_train_tensor.numpy()

class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)

# Initialize Model, Function And Optimizer
model_hybrid_rnn = HybridRNN(vocab_size=len(word2idx), embedding_dim=100, hidden
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_hybrid_rnn.parameters(), lr=learning_rate)
```

In [178…
```
# Creating The Training Loop
for epoch in range(epochs):
    model_hybrid_rnn.train()
    total_loss = 0
    for texts, structured, labels in train_loader_hybrid:
        texts = texts.to(device)
        structured = structured.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_hybrid_rnn(texts, structured)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader_hybrid)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.5956
Epoch [2/10] - Loss: 0.5023
Epoch [3/10] - Loss: 0.4801
Epoch [4/10] - Loss: 0.4686
Epoch [5/10] - Loss: 0.4615
Epoch [6/10] - Loss: 0.4537
Epoch [7/10] - Loss: 0.4456
Epoch [8/10] - Loss: 0.4393
Epoch [9/10] - Loss: 0.4540
Epoch [10/10] - Loss: 0.4499
```

In [179…

```
# Evaluating The Hybrid RNN Model
def evaluate_hybrid_rnn(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in val_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("Hybrid RNN Validation Performance")
print("---------------------------------")
score = evaluate_hybrid_rnn(model_hybrid_rnn, val_loader_hybrid)
print("F1-score:", score)
```

```
Hybrid RNN Validation Performance
---------------------------------
F1-score: 0.769
```

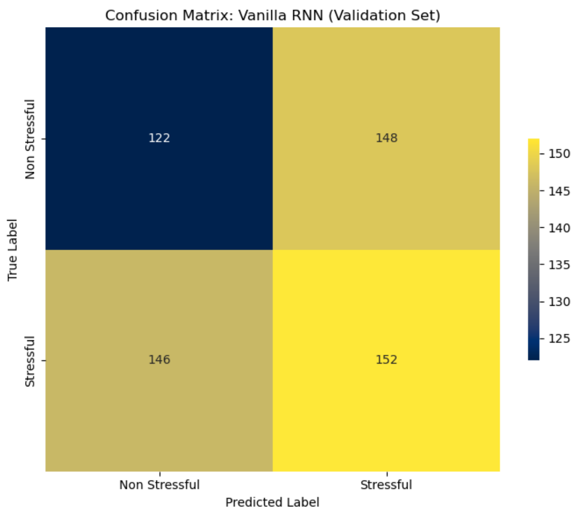
The **Hybrid RNN** achieved an F1-score of **0.7690**, representing a substantial improvement over the Vanilla RNN model. This increase demonstrates that incorporating structured features alongside text allows the model to capture more informative signals, similar to the performance trends observed in the CNN and Hybrid CNN models. By leveraging both textual and numerical features, the Hybrid RNN reduces false positives and false negatives compared to the text-only RNN, leading to a more balanced classification of stressful and non-stressful texts.

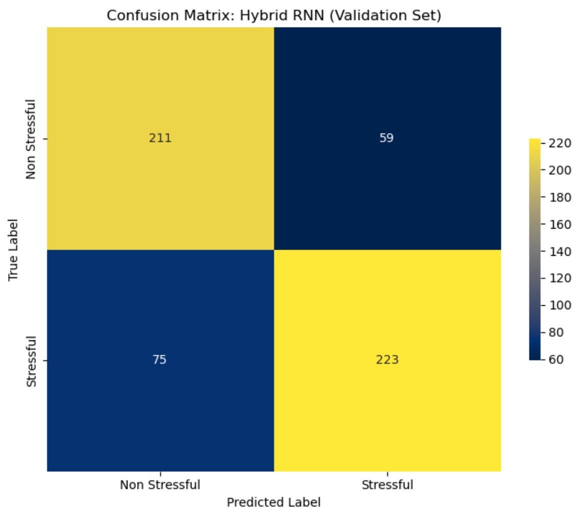
However, despite this improvement, the **Hybrid RNN still attains a lower F1-score than the Hybrid CNN model**. This suggests that the CNN architecture is more effective than the RNN at extracting the key patterns and features needed for stress classification in this dataset. CNNs ability to efficiently capture local n-gram patterns in text may provide a stronger representation for distinguishing stressful from non-stressful content, making them better suited for this particular classification task.

Now let us explore the confusion matrix of these two CNN models and explore the differences:

In [181…
```
# Implementing Prediction functions
def get_text_predictions_rnn(model, data_loader):
    """Predictions for Vanilla RNN (text only)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in data_loader:
            texts = texts.to(device)
            logits = model(texts)  # assume model outputs logits
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

def get_hybrid_predictions_rnn(model, data_loader):
    """Predictions for Hybrid RNN (text + structured features)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in data_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

# Get Predictions
y_val_labels_vanilla_rnn, y_val_pred_vanilla_rnn = get_text_predictions_rnn(mode
y_val_labels_hybrid_rnn, y_val_pred_hybrid_rnn = get_hybrid_predictions_rnn(mode
```

In [182…
```
# Compute confusion matrices
cm_vanilla_rnn = confusion_matrix(y_val_labels_vanilla_rnn, y_val_pred_vanilla_r
cm_hybrid_rnn = confusion_matrix(y_val_labels_hybrid_rnn, y_val_pred_hybrid_rnn)

cms = [cm_vanilla_rnn, cm_hybrid_rnn]
titles = ['Vanilla RNN', 'Hybrid RNN']
class_names = ['Non Stressful', 'Stressful']

# Plot
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
for ax, cm, title in zip(axes, cms, titles):
    sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap='cividis',
                xticklabels=class_names, yticklabels=class_names,
                cbar=True, cbar_kws={'shrink':0.5})
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('True Label')
    ax.set_title(f'Confusion Matrix: {title} (Validation Set)')

plt.tight_layout()
plt.show()
```





Key Observations:

Vanilla RNN:

- Correct predictions: 122 Non-Stressful, 152 Stressful
- Misclassifications: 148 Non-Stressful, 146 Stressful
- Very high error rates in both directions, showing difficulty in distinguishing stress-related language patterns

Hybrid RNN:

- Correct predictions: 211 Non-Stressful, 223 Stressful
- Misclassifications: 59 Non-Stressful, 75 Stressful
- Much stronger performance, with significantly fewer errors compared to Vanilla RNN

Overall Accuracy:

Hybrid RNN reduces false alarms by more than half as compared to Vanilla RNN. Hybrid RNN also reduces missed stressful cases substantially as compared to Vanilla RNN. Vanilla RNN struggles equally with both classes (high confusion both ways). Hybrid RNN shows better separation, suggesting improved feature extraction and contextual understanding.

Possible Insights:

Vanilla RNN struggles with long-range dependencies in text, leading to confusion between stressful and non-stressful posts. Stress cues often depend on context spread across sentences, which simple RNNs fail to capture. Hybrid RNN integrates additional mechanisms (attention layers, bidirectional processing or hybrid embeddings), enabling it to better capture nuanced stress signals. For stress detection in social media, Hybrid RNN is far more reliable, reducing both false alarms and missed detections. This is critical for applications like mental health monitoring, where both types of errors can have serious consequences.

Now let us proceed with the next RNN variant model, **LSTM**.

Method 2 Part 2: LSTM

Long Short-Term Memory networks (LSTMs) are an advanced type of Recurrent Neural Network designed to address the limitations of Vanilla RNNs, particularly their difficulty in capturing long-term dependencies due to the vanishing gradient problem. Unlike a standard RNN, which maintains a single hidden state that is updated at every time step, LSTMs introduce a memory cell along with gating mechanisms: namely the input gate, forget gate and output gate that carefully control the flow of information. These gates allow the network to retain important information over long sequences while selectively forgetting irrelevant or redundant data.

In the context of stress classification from text, this capability is particularly valuable because the model can remember relevant emotional or contextual cues from earlier words in a sentence, paragraph or post that may indicate stress. By effectively modeling both short-term and long-term dependencies in the text, LSTMs are better equipped than Vanilla RNNs to capture subtle linguistic patterns and provide more accurate predictions of whether a text is stressful or non-stressful.

LSTM Model Pipeline:

Step 1: Create Dataset And DataLoader

We first split the Dreaddit dataset into training and validation sets, and create PyTorch DataLoaders. Only the Text column is used here. The labels are converted to tensors for training.

In [185…
```python
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(0)

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

In [186…
```python
# Split into train/validation (80/20)
train_df, val_df = train_test_split(dreaddit_train, test_size=0.2, random_state=

# Convert text and labels to tensors
y_train_tensor = torch.tensor(train_df['Label'].values, dtype=torch.float)
y_val_tensor = torch.tensor(val_df['Label'].values, dtype=torch.float)
```

```
train_dataset = TensorDataset(X_train_padded, y_train_tensor)
val_dataset = TensorDataset(X_val_padded, y_val_tensor)

# Setting The Hyperparameter Values
batch_size = 32

# Deterministic shuffling
g = torch.Generator()
g.manual_seed(0)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, ge
val_loader = DataLoader(val_dataset, batch_size=batch_size,shuffle=False)
```

Step 2: Define The LSTM Model

The LSTM model includes:

1. An embedding layer to convert token indices to dense vectors.
2. An LSTM layer that maintains hidden and cell states for capturing long-term dependencies.
3. A fully connected output layer with a sigmoid activation for binary classification.

Compared to Vanilla RNNs, LSTM's gating mechanism allows the model to selectively retain or forget information, which is particularly useful for identifying subtle stress cues across a sequence of text.

In [188…

```
# Defining The LSTM Model
class LSTM_Text(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(LSTM_Text, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            batch_first=True
        )
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(hidden_dim, 1)

    def forward(self, text):
        embedded = self.embedding(text)
        output, (hidden, cell) = self.lstm(embedded)
        hidden = self.dropout(hidden[-1])
        logits = self.fc(hidden)
        return logits
```

Step 3: Initialize Model, Loss And Optimizer

We instantiate the LSTM model and move it to the appropriate device. Binary Cross-Entropy is used as the loss function for binary classification, and Adam is used as the optimizer.

In [190…

```
# Setting The Hyperparameter Values
embedding_dim = 100
```

```
hidden_dim = 128
learning_rate = 0.001
epochs = 10

# Obtaining Class Weights - Handle Class Imbalance
y_train_np = y_train_tensor.numpy()
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)

# Initialize Model, Function And Optimizer
model_lstm = LSTM_Text(vocab_size=len(word2idx), embedding_dim=embedding_dim, hi
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_lstm.parameters(), lr=learning_rate)
```

Step 4: Model Training

During training, the model processes mini-batches of text sequences, computes predictions, calculates loss and updates the model weights via backpropagation. We also print the average loss per epoch to monitor convergence.

In [192…

```
# Creating The Training Loop
for epoch in range(epochs):
    model_lstm.train()
    total_loss = 0
    for texts, labels in train_loader:
        texts = texts.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_lstm(texts)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.6778
Epoch [2/10] - Loss: 0.6753
Epoch [3/10] - Loss: 0.6741
Epoch [4/10] - Loss: 0.6734
Epoch [5/10] - Loss: 0.6729
Epoch [6/10] - Loss: 0.6733
Epoch [7/10] - Loss: 0.6720
Epoch [8/10] - Loss: 0.6778
Epoch [9/10] - Loss: 0.6729
Epoch [10/10] - Loss: 0.6723
```

Step 5: Evaluating The Model

This function evaluates the model on the validation set. Predictions are thresholded at 0.5 to obtain binary labels and the F1-score is computed to measure performance, balancing precision and recall.

In [194…

```
# Evaluating The LSTM Model
def evaluate_lstm(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in val_loader:
            texts = texts.to(device)
            logits = model(texts)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("\nLSTM Validation Performance")
print("-----------------------------")
score = evaluate_lstm(model_lstm, val_loader)
print("F1-score:", score)
```

```
LSTM Validation Performance
-----------------------------
F1-score: 0.6867
```

The **LSTM model** achieved an F1-score of **0.6867**, which is higher than the Vanilla RNN but still notably lower than the CNN model. While LSTMs are designed to capture long-term dependencies and contextual information in sequential data, their performance in this case suggests that they may not be as effective as CNNs at identifying the local patterns and key n-gram features that are particularly informative for classifying stressful versus non-stressful text.

Additionally, like the RNN, this LSTM model was trained using only the Text column, without incorporating structured features such as LIWC-derived numerical variables. The exclusion of these additional features likely limits the model's ability to leverage important linguistic and psychological cues, resulting in a lower overall F1-score compared to CNN-based architectures that either focus on local textual patterns (CNN) or integrate both text and structured features (hybrid CNN).

Now we will implement the **hybrid LSTM model** using the similar approach as hybrid CNN and RNN.

In [196…

```
# Implmenting The Hybrid LSTM Model
class HybridLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_structured_fea
        super(HybridLSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim, ba
        self.text_dropout = nn.Dropout(0.5)
        self.struct_fc = nn.Linear(num_structured_features, 64)
        self.final_fc = nn.Linear(hidden_dim + 64, 1)

    def forward(self, text, structured):
        embedded = self.embedding(text)
        output, (hidden, cell) = self.lstm(embedded)
        text_features = self.text_dropout(hidden[-1])
        structured_features = F.relu(self.struct_fc(structured))
        combined = torch.cat((text_features, structured_features), dim=1)
        logits = self.final_fc(combined)
        return logits
```

In [197…
```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(0)

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

In [198…
```
# Setting The Hyperparameter Values
batch_size = 32

# Convert text and labels to tensors
train_dataset_hybrid = TensorDataset(X_train_padded, structured_train_tensor, y_
val_dataset_hybrid = TensorDataset(X_val_padded, structured_val_tensor, y_val_te

# Deterministic shuffling
g = torch.Generator()
g.manual_seed(0)

train_loader_hybrid = DataLoader(train_dataset_hybrid, batch_size=batch_size, sh
val_loader_hybrid = DataLoader(val_dataset_hybrid, batch_size=batch_size, shuffl
```

In [199…
```
# Setting The Hyperparameter Values
embedding_dim = 100
hidden_dim = 128
learning_rate = 0.001
epochs = 10

# Obtaining Class Weights - Handle Class Imbalance
y_train_np = y_train_tensor.numpy()
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)

# Initialize Model, Function And Optimizer
model_hybrid_lstm = HybridLSTM(vocab_size=len(word2idx), embedding_dim=embedding
                               num_structured_features=structured_train_tensor.s
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_hybrid_lstm.parameters(), lr=learning_rate)
```

In [200…
```
# Training The LSTM Hybrid Model
for epoch in range(epochs):
    model_hybrid_lstm.train()
    total_loss = 0
    for texts, structured, labels in train_loader_hybrid:
        texts = texts.to(device)
        structured = structured.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_hybrid_lstm(texts, structured)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader_hybrid)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.5999
Epoch [2/10] - Loss: 0.5053
Epoch [3/10] - Loss: 0.4814
Epoch [4/10] - Loss: 0.4691
Epoch [5/10] - Loss: 0.4619
Epoch [6/10] - Loss: 0.4517
Epoch [7/10] - Loss: 0.4436
Epoch [8/10] - Loss: 0.4364
Epoch [9/10] - Loss: 0.4290
Epoch [10/10] - Loss: 0.4228
```

In [201…

```
# Evaluating The LSTM Hybrid Model
def evaluate_hybrid_lstm(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in val_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("\nHybrid LSTM Validation Performance")
print("------------------------------------")
score = evaluate_hybrid_lstm(model_hybrid_lstm, val_loader_hybrid)
print("F1-score:", score)
```

```
Hybrid LSTM Validation Performance
------------------------------------
F1-score: 0.769
```

The **Hybrid LSTM** achieved an F1-score of **0.7690**, which is similar to the F1-score of the Hybrid RNN.

However, the Hybrid LSTM still performs lower than the Hybrid CNN model. This difference may be due to the CNN's superior ability to capture local n-gram patterns and salient textual features that are particularly informative for stress detection. While LSTMs are effective at modeling sequential dependencies, they may be less efficient at detecting these localized patterns across many posts, especially in short to medium-length text where critical stress indicators are often captured by local combinations of words rather than long-range dependencies.

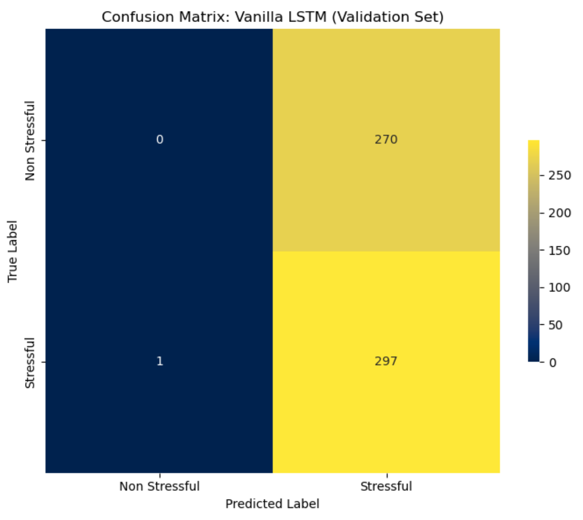
Now let us explore the confusion matrix of these two LSTM models and explore the differences:

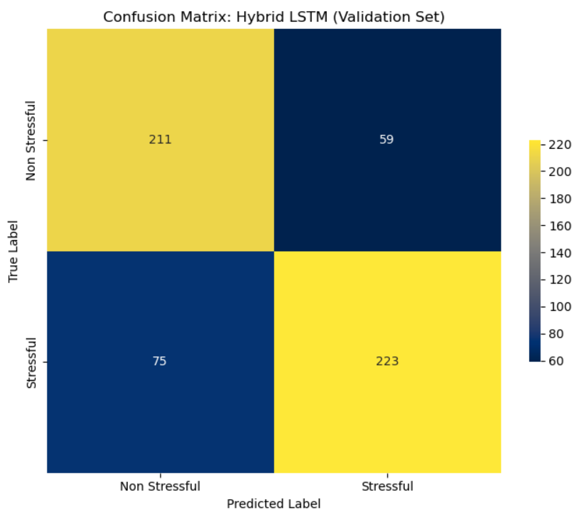
In [203…

```
# Implementing Prediction functions
def get_text_predictions_lstm(model, data_loader):
    """Predictions for Vanilla LSTM (text only)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in data_loader:
            texts = texts.to(device)
            logits = model(texts)  # Logits output
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

def get_hybrid_predictions_lstm(model, data_loader):
    """Predictions for Hybrid LSTM (text + structured)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in data_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

# Get predictions
y_val_labels_vanilla_lstm, y_val_pred_vanilla_lstm = get_text_predictions_lstm(m
y_val_labels_hybrid_lstm, y_val_pred_hybrid_lstm = get_hybrid_predictions_lstm(m
```

In [204…

```
# Compute confusion matrices
cm_vanilla_lstm = confusion_matrix(y_val_labels_vanilla_lstm, y_val_pred_vanilla
cm_hybrid_lstm = confusion_matrix(y_val_labels_hybrid_lstm, y_val_pred_hybrid_ls

cms = [cm_vanilla_lstm, cm_hybrid_lstm]
titles = ['Vanilla LSTM', 'Hybrid LSTM']
class_names = ['Non Stressful', 'Stressful']

# Plot
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
for ax, cm, title in zip(axes, cms, titles):
    sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap='cividis',
                xticklabels=class_names, yticklabels=class_names,
                cbar=True, cbar_kws={'shrink':0.5})
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('True Label')
    ax.set_title(f'Confusion Matrix: {title} (Validation Set)')

plt.tight_layout()
plt.show()
```





Key Observations:

Vanilla LSTM:

- Correct predictions: 0 Non-Stressful, 297 Stressful
- Misclassifications: 270 Non-Stressful, 1 Stressful
- Essentially predicts almost everything as Stressful, showing severe class imbalance in its outputs

Hybrid LSTM:

- Correct predictions: 211 Non-Stressful, 223 Stressful
- Misclassifications: 59 Non-Stressful, 75 Stressful
- Much stronger performance, with balanced recognition of both classes

Overall Accuracy:

Hybrid LSTM is significantly more accurate than Vanilla LSTM. Hybrid LSTM avoids the "everything is stressful" bias. Vanilla LSTM almost never misses stressful posts, but at the cost of overwhelming false alarms. Vanilla LSTM is heavily skewed toward Stressful predictions, failing to capture Non-Stressful cases. Hybrid LSTM achieves better balance, recognizing both classes more reliably.

Possible Insights:

Vanilla LSTM shows mode collapse: it defaults to predicting Stressful posts, likely due to overfitting or imbalance in training signals. This makes it unusable in practice since it cannot distinguish normal posts. Hybrid LSTM integrates richer mechanisms (eg. attention, bidirectional layers or hybrid embeddings), allowing it to capture nuanced differences between stressful and non-stressful language. Vanilla LSTM may be useful if the priority is never missing stressful posts, but it generates too many false alarms. Hybrid LSTM is far more practical, striking a balance between sensitivity and specificity, making it suitable for real-world stress detection systems.

Now we will look at the next RNN variant model, **GRU**.

Method 2 Part 3: GRU

A Gated Recurrent Unit (GRU) is a type of recurrent neural network designed to address the vanishing gradient problem found in Vanilla RNNs while maintaining a simpler structure than LSTMs. The GRU combines the memory cell and hidden state into a single representation and uses only two gating mechanisms: an update gate and a reset gate. The update gate controls how much past information should be retained while the reset gate determines how much previous information should be forgotten when incorporating new input. Compared to LSTM, GRU has fewer parameters, making it computationally more efficient and often faster to train, while still being capable of modeling long-term dependencies in sequential data.

For the Dreaddit dataset, which involves classifying text as stressful or non-stressful, a GRU can effectively capture contextual and emotional patterns across sequences of words without the complexity of a full LSTM. Since stress-related cues may depend on both immediate word combinations and broader sentence context, the GRU's gating mechanism allows it to selectively retain meaningful emotional signals while filtering out irrelevant content. Additionally, because GRUs are generally less prone to overfitting on smaller datasets due to their simpler architecture, they can offer a strong balance between performance and efficiency for stress detection tasks.

GRU Model Pipline:

Step 1: Prepare Dataset And DataLoader

We prepare the training and validation datasets using padded text sequences (`X_train_padded` , `X_val_padded`) and their corresponding labels. These are wrapped into TensorDataset objects and loaded into DataLoader for mini-batch training. Deterministic shuffling ensures reproducible experiments.

NOTE: In this step, We also fix the random seed to ensure reproducibility. This guarantees consistent weight initialization, dataset shuffling and model behavior across multiple runs, which is essential when comparing F1-scores across RNN, LSTM, CNN, and GRU models.

In [207…

```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(0)

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

In [208…

```
# Setting The Hyperparameter Values
batch_size = 32
```

```
# Convert text and labels to tensors
train_dataset = TensorDataset(X_train_padded, y_train_tensor)
val_dataset = TensorDataset(X_val_padded, y_val_tensor)

# Deterministic shuffling
g = torch.Generator()
g.manual_seed(0)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, ge
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

Step 2: Define The GRU Model

In this step, we define the GRU architecture. The model consists of:

1. Embedding Layer - Converts word indices into dense vector representations.

2. GRU Layer - Captures sequential dependencies using update and reset gates.

3. Dropout Layer - Reduces overfitting.

4. Fully Connected Layer - Produces the final binary classification output.

Unlike LSTM, GRU combines the cell state and hidden state into a single representation, making it computationally simpler while still retaining long-term contextual information.

In [210…
```
# Implementing The GRU Model
class GRU_Text(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(GRU_Text, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.gru = nn.GRU(input_size=embedding_dim, hidden_size=hidden_dim, batc
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(hidden_dim, 1)

    def forward(self, text):
        embedded = self.embedding(text)
        output, hidden = self.gru(embedded)
        hidden = self.dropout(hidden[-1])
        logits = self.fc(hidden)
        return logits
```

Step 3: Initialize Model, Loss Function And Optimizer

We now instantiate the GRU model and move it to the appropriate device (CPU/GPU). We use Binary Cross-Entropy loss for binary classification and Adam optimizer for efficient gradient updates.

In [212…
```
# Setting The Hyperparameter Values
embedding_dim = 100
hidden_dim = 128
learning_rate = 0.001
epochs = 10

# Compute Class Weights - Handle Class Imbalance
y_train_np = y_train_tensor.numpy()
```

```
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)

# Initialize Model, Function And Optimizer
model_gru = GRU_Text(vocab_size=len(word2idx), embedding_dim=embedding_dim, hidd
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_gru.parameters(), lr=learning_rate)
```

Step 4: Model Training

During training, the model processes mini-batches of text data, computes predictions, calculates loss, performs backpropagation, and updates weights. We track average loss per epoch to monitor learning progress.

In [214…

```
# Training The GRU Model
for epoch in range(epochs):
    model_gru.train()
    total_loss = 0
    for texts, labels in train_loader:
        texts = texts.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_gru(texts)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.6779
Epoch [2/10] - Loss: 0.6749
Epoch [3/10] - Loss: 0.6742
Epoch [4/10] - Loss: 0.6740
Epoch [5/10] - Loss: 0.6729
Epoch [6/10] - Loss: 0.6733
Epoch [7/10] - Loss: 0.6721
Epoch [8/10] - Loss: 0.6724
Epoch [9/10] - Loss: 0.6713
Epoch [10/10] - Loss: 0.6724
```

Step 5: Evaluating The Model

To assess performance, we switch the model to evaluation mode and compute predictions on the validation set. Predictions are thresholded at 0.5 to generate binary outputs and the F1-score is calculated to balance precision and recall.

In [216…

```
# Evaluating The GRU Model
def evaluate_gru(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in val_loader:
            texts = texts.to(device)
            logits = model(texts)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("\nGRU Validation Performance")
print("----------------------------")
score = evaluate_gru(model_gru, val_loader)
print("F1-score:", score)
```

```
GRU Validation Performance
----------------------------
F1-score: 0.6852
```

The **GRU model** achieved an F1-score of **0.6852**, which is very similar to the LSTM model and higher than the Vanilla RNN, indicating a slight improvement over the basic recurrent architecture. This suggests that the gating mechanism in GRU, specifically the update and reset gates, helps mitigate the vanishing gradient problem and allows the model to retain more relevant contextual information compared to a standard RNN.

However, despite this improvement, the GRU still performs noticeably lower than the CNN model. This may indicate that stress detection in the Dreaddit dataset relies more heavily on capturing local lexical patterns and key n-grams, which CNNs are particularly effective at identifying.

While GRUs are efficient and capable of modeling sequential dependencies, **they may not provide a substantial advantage over LSTM in this specific task** and both recurrent architectures appear less effective than CNN-based models for classifying stressful versus non-stressful text in this dataset.

Now we will implement the **hybrid GRU model** using the similar approach as hybrid CNN, RNN and LSTM models.

In [218…
```
# Implmenting The Hybrid GRU Model
class HybridGRU(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_structured_fea
        super(HybridGRU, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.gru = nn.GRU(input_size=embedding_dim, hidden_size=hidden_dim, batc
        self.text_dropout = nn.Dropout(0.5)
        self.struct_fc = nn.Linear(num_structured_features, 64)
        self.final_fc = nn.Linear(hidden_dim + 64, 1)

    def forward(self, text, structured):
        embedded = self.embedding(text)
        output, hidden = self.gru(embedded)
        text_features = self.text_dropout(hidden[-1])
        structured_features = F.relu(self.struct_fc(structured))
        combined = torch.cat((text_features, structured_features), dim=1)
        logits = self.final_fc(combined)
        return logits
```

In [219…
```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(0)

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

In [220…

```
# Setting The Hyperparameter Values
batch_size = 32

# Convert text and labels to tensors
train_dataset_hybrid = TensorDataset(X_train_padded, structured_train_tensor, y_
val_dataset_hybrid = TensorDataset(X_val_padded, structured_val_tensor, y_val_te

# Deterministic shuffling
g = torch.Generator()
g.manual_seed(0)

train_loader_hybrid = DataLoader(train_dataset_hybrid, batch_size=batch_size, sh
val_loader_hybrid = DataLoader(val_dataset_hybrid, batch_size=batch_size, shuffl
```

In [221…

```
# Setting The Hyperparameter Values
embedding_dim = 100
hidden_dim = 128
learning_rate = 0.001
epochs = 10

# Compute Class Weights - Handle Class Imbalance
y_train_np = y_train_tensor.numpy()
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)

# Initialize Model, Function And Optimizer
model_hybrid_gru = HybridGRU(vocab_size=len(word2idx), embedding_dim=embedding_d
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_hybrid_gru.parameters(), lr=learning_rate)
```

In [222…

```
# Training The Hybrid GRU Model
for epoch in range(epochs):
    model_hybrid_gru.train()
    total_loss = 0
    for texts, structured, labels in train_loader_hybrid:
        texts = texts.to(device)
        structured = structured.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_hybrid_gru(texts, structured)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader_hybrid)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.6073
Epoch [2/10] - Loss: 0.5118
Epoch [3/10] - Loss: 0.4860
Epoch [4/10] - Loss: 0.4737
Epoch [5/10] - Loss: 0.4631
Epoch [6/10] - Loss: 0.4555
Epoch [7/10] - Loss: 0.4529
Epoch [8/10] - Loss: 0.4431
Epoch [9/10] - Loss: 0.4355
Epoch [10/10] - Loss: 0.4310
```

In [223…

```
# Evaluating The Hybrid GRU Model
def evaluate_hybrid_gru(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in val_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("\nHybrid GRU Validation Performance")
print("-----------------------------------")
score = evaluate_hybrid_gru(model_hybrid_gru, val_loader_hybrid)
print("F1-score:", score)
```

```
Hybrid GRU Validation Performance
-----------------------------------
F1-score: 0.7676
```

The **Hybrid GRU model** achieved an F1-score of **0.7676**, which is slightly lower than both the Hybrid RNN and Hybrid LSTM models, indicating that while the GRU architecture is effective at capturing contextual dependencies in text, it may not fully leverage the structured features in the Dreaddit dataset as well as the other recurrent models. The GRU's update and reset gates still allow it to efficiently model sequential information with fewer parameters than LSTM, potentially reducing overfitting, but in this case, the simpler RNN or the more expressive LSTM may have better generalized to the task.

When integrated with structured features, the model benefits from both sequential linguistic information and psychologically meaningful numerical signals, but the performance suggests that the balance between sequential modeling and structured feature integration is slightly more favorable in the Hybrid RNN and Hybrid LSTM.

The **Hybrid GRU also performs lower than the Hybrid CNN**. This suggests that, for the Dreaddit dataset, capturing local lexical patterns and key n-grams, an area where CNNs excel, remains more critical than modeling long-range sequential dependencies.

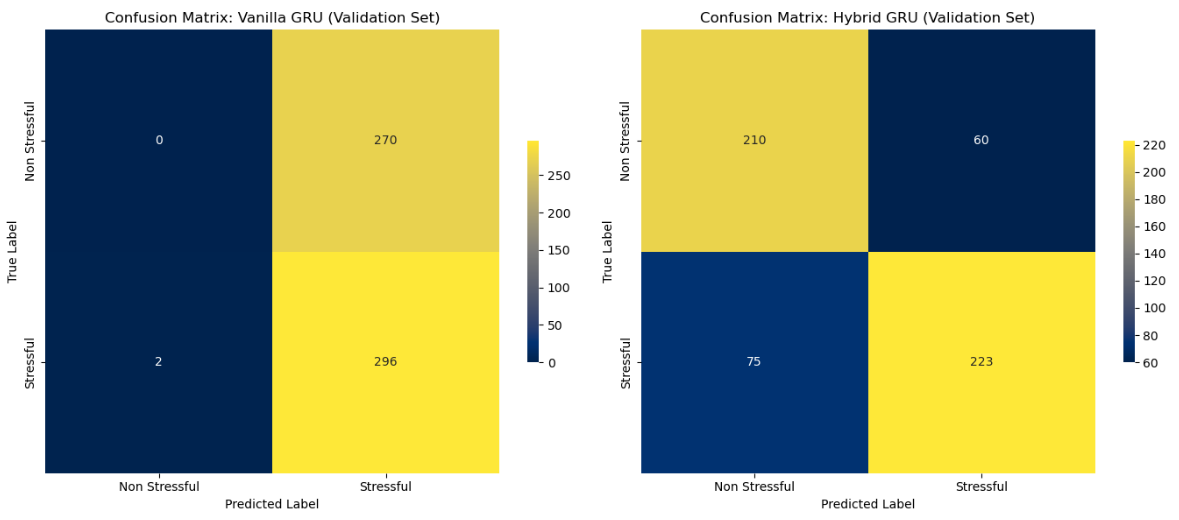
Now let us explore the confusion matrix of these two GRU models and explore the differences:

In [225…

```
# Prediction functions
def get_text_predictions_gru(model, data_loader):
    """Predictions for Vanilla GRU (text only)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in data_loader:
            texts = texts.to(device)
            logits = model(texts)  # output logits
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

def get_hybrid_predictions_gru(model, data_loader):
    """Predictions for Hybrid GRU (text + structured features)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in data_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

# Get predictions
y_val_labels_vanilla_gru, y_val_pred_vanilla_gru = get_text_predictions_gru(mode
y_val_labels_hybrid_gru, y_val_pred_hybrid_gru = get_hybrid_predictions_gru(mode
```

In [226…

```
# Compute confusion matrices
cm_vanilla_gru = confusion_matrix(y_val_labels_vanilla_gru, y_val_pred_vanilla_g
cm_hybrid_gru = confusion_matrix(y_val_labels_hybrid_gru, y_val_pred_hybrid_gru)

cms = [cm_vanilla_gru, cm_hybrid_gru]
titles = ['Vanilla GRU', 'Hybrid GRU']
class_names = ['Non Stressful', 'Stressful']

# Plot confusion matrices
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
for ax, cm, title in zip(axes, cms, titles):
    sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap='cividis',
                xticklabels=class_names, yticklabels=class_names,
                cbar=True, cbar_kws={'shrink':0.5})
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('True Label')
    ax.set_title(f'Confusion Matrix: {title} (Validation Set)')

plt.tight_layout()
plt.show()
```



Key Observations:

Vanilla GRU:

- Correct predictions: 0 Non-Stressful, 296 Stressful
- Misclassifications: 270 Non-Stressful, 2 Stressful
- Similar to Vanilla LSTM, it collapses into predicting almost everything as Stressful, failing to capture Non-Stressful posts

Hybrid GRU:

- Correct predictions: 210 Non-Stressful, 223 Stressful
- Misclassifications: 60 Non-Stressful, 75 Stressful
- Much stronger performance, with balanced recognition across both classes

Overall Accuracy:

Hybrid GRU is significantly more accurate than Vanilla GRU. Hybrid GRU avoids the "everything is stressful" bias. Vanilla GRU almost never misses stressful posts, but at the cost of overwhelming false alarms. Vanilla GRU is heavily skewed toward Stressful predictions, failing to recognize Non-Stressful cases. Hybrid GRU achieves better balance, recognizing both classes more reliably.

Possible Insights:

Vanilla GRU suffers from class bias collapse, similar to Vanilla LSTM, defaulting to Stressful predictions. This suggests difficulty in learning discriminative features for Non-Stressful posts. Hybrid GRU integrates richer mechanisms (likely attention, bidirectional layers or hybrid embeddings), enabling it to capture nuanced differences between stressful and non-stressful language. Vanilla GRU may be useful if the priority is never missing stressful posts, but it generates too many false alarms. Hybrid GRU is far more practical, striking a balance between sensitivity and specificity, making it suitable for real-world stress detection systems.

Now let us move to the last deep learning model being considered, **Bidirectional LSTM**.

Deep Learning Method 3: Bidirectional LSTM

A **Bidirectional LSTM (BiLSTM)** is an extension of the standard LSTM architecture that processes a sequence in both forward and backward directions. While a traditional LSTM reads text from the first word to the last (left -> right), a Bidirectional LSTM consists of two LSTM layers:

- One processes the sequence forward
- The other processes the sequence backward

The outputs from both directions are then concatenated to form a richer representation of the text. This means that each word's representation is informed not only by the words that come before it, but also by the words that come after it. In natural language processing, this is particularly powerful because the meaning of a word often depends on its surrounding context from both directions.

For the Dreaddit stress classification dataset, Bidirectional LSTM is especially suitable because stress-related signals may depend heavily on contextual nuances. For example, phrases such as “I thought I was fine, but now I'm overwhelmed” require understanding both earlier and later parts of the sentence to correctly interpret emotional intensity. A unidirectional model may partially capture this, but a BiLSTM can better model the full semantic flow of the text.

Additionally, stress indicators are often subtle and context-dependent. Negations, contrast words ("but", "however") or temporal shifts can significantly change emotional meaning. Since BiLSTM captures information from both past and future tokens, it can generate more context-aware representations of posts, potentially improving classification performance.

Bidirectional LSTM Model Pipeline

Step 1: Prepare Dataset And DataLoader

In this step, we prepare the padded text sequences and corresponding labels for training and validation. The data is wrapped into TensorDataset objects and loaded using DataLoader to enable mini-batch training.

In [229…
```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(0)

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

In [230…
```
# Setting The Hyperparameter Values
```

```
batch_size = 32

# Convert text and labels to tensors
train_dataset = TensorDataset(X_train_padded, y_train_tensor)
val_dataset = TensorDataset(X_val_padded, y_val_tensor)

# Deterministic shuffling
g = torch.Generator()
g.manual_seed(0)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, ge
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

Step 2: Define The Bidirectional LSTM Model

A Bidirectional LSTM consists of two LSTMs:

- One processes the sequence forward
- One processes the sequence backward

Their hidden states are concatenated to produce a richer contextual representation.
Since the hidden dimension doubles (forward + backward), the final fully connected layer must account for this.

The architecture includes:

- Embedding layer
- Bidirectional LSTM layer
- Dropout layer
- Fully connected output layer

In [232…

```
# Implementing The Bidirectional LSTM Model
class BiLSTM_Text(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(BiLSTM_Text, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim, ba
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(hidden_dim * 2, 1)

    def forward(self, text):
        embedded = self.embedding(text)
        output, (hidden, cell) = self.lstm(embedded)
        hidden_forward = hidden[-2]
        hidden_backward = hidden[-1]
        hidden_combined = torch.cat((hidden_forward, hidden_backward), dim=1)
        hidden_combined = self.dropout(hidden_combined)
        logits = self.fc(hidden_combined)
        return logits
```

Step 3: Initialize Model, Loss Function And Optimizer

We initialize the BiLSTM model and move it to the selected device. Binary Cross-Entropy loss is used for classification, and Adam optimizer is selected for efficient parameter updates.

In [234…
```
# Setting The Hyperparameter Values
embedding_dim = 100
hidden_dim = 128
learning_rate = 0.001
epochs = 10

# Obtaining Model Weights - Handle Class Imbalance
y_train_np = y_train_tensor.numpy()
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)

# Initialize Model, Function And Optimizer
model_bilstm = BiLSTM_Text(vocab_size=len(word2idx), embedding_dim=embedding_dim
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_bilstm.parameters(), lr=learning_rate)
```

Step 4: Model Training

The model is trained over multiple epochs using mini-batches. For each batch:

- Forward pass generates predictions
- Loss is computed
- Backpropagation updates weights

We monitor the average training loss per epoch.

In [236…
```
# Training The Bi-Directional LSTM Model
for epoch in range(epochs):
    model_bilstm.train()
    total_loss = 0
    for texts, labels in train_loader:
        texts = texts.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_bilstm(texts)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.6663
Epoch [2/10] - Loss: 0.6185
Epoch [3/10] - Loss: 0.5717
Epoch [4/10] - Loss: 0.5065
Epoch [5/10] - Loss: 0.5411
Epoch [6/10] - Loss: 0.4965
Epoch [7/10] - Loss: 0.4258
Epoch [8/10] - Loss: 0.3753
Epoch [9/10] - Loss: 0.3160
Epoch [10/10] - Loss: 0.4073
```

Step 5: Evaluating The Model

We evaluate the model on the validation set by switching to evaluation mode. Predictions are thresholded at 0.5 to obtain binary labels, and the F1-score is computed to balance precision and recall.

In [238…

```
# Evaluating The Bidirectional LSTM Model
def evaluate_bilstm(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in val_loader:
            texts = texts.to(device)
            logits = model(texts)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("\nBidirectional LSTM Validation Performance")
print("-------------------------------------------")
score = evaluate_bilstm(model_bilstm, val_loader)
print("F1-score:", score)
```

```
Bidirectional LSTM Validation Performance
-------------------------------------------
F1-score: 0.7183
```

The **Bidirectional LSTM** achieved an F1-score of **0.7183**, which is considerably higher than the Vanilla RNN, LSTM and GRU models. This improvement suggests that incorporating contextual information from both forward and backward directions enables the model to better capture nuanced emotional cues within the Dreaddit posts. Since stress-related expressions often depend on how earlier and later parts of a sentence interact, such as contrastive statements or delayed emotional revelations, the bidirectional structure provides a richer representation of the text compared to unidirectional recurrent models.

However, despite this improvement over other recurrent architectures, the **Bidirectional LSTM still performs lower than the Vanilla CNN**. This indicates that for the Dreaddit dataset detecting strong local lexical patterns and key n-grams remains more impactful than modeling long-range contextual dependencies. While BiLSTM enhances sequential understanding, CNN-based models appear more effective at extracting the salient stress-indicative features necessary for optimal classification performance in this task.

Now we will implement the **hybrid bidirectional LSTM model** using the similar approach as hybrid CNN, RNN and GRU models.

In [240…

```
# Implementing The Hybrid Bidirectional LSTM Model
class HybridBiLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_structured_fea
        super(HybridBiLSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_dim, ba
        self.text_dropout = nn.Dropout(0.5)
        self.struct_fc = nn.Linear(num_structured_features, 64)
        self.final_fc = nn.Linear(hidden_dim * 2 + 64, 1)

    def forward(self, text, structured):
        embedded = self.embedding(text)
        output, (hidden, cell) = self.lstm(embedded)
        hidden_forward = hidden[-2]
        hidden_backward = hidden[-1]
        text_features = torch.cat((hidden_forward, hidden_backward), dim=1)
        text_features = self.text_dropout(text_features)
        structured_features = F.relu(self.struct_fc(structured))
        combined = torch.cat((text_features, structured_features), dim=1)
        logits = self.final_fc(combined)
        return logits
```

In [241…

```
# Setting A Seed (To Ensure No Changes In F1 Score When Code Is Run)
def set_seed(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(0)

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

In [242…

```
# Setting The Hyperparameter Values
batch_size = 32

# Convert text and labels to tensors
train_dataset_hybrid = TensorDataset(X_train_padded, structured_train_tensor, y_
val_dataset_hybrid = TensorDataset(X_val_padded, structured_val_tensor, y_val_te

# Deterministic shuffling
g = torch.Generator()
g.manual_seed(0)

train_loader_hybrid = DataLoader(train_dataset_hybrid, batch_size=batch_size, sh
val_loader_hybrid = DataLoader(val_dataset_hybrid, batch_size=batch_size, shuffl
```

In [243…

```
# Setting The Hyperparameter Values
embedding_dim = 100
hidden_dim = 128
learning_rate = 0.001
epochs = 10

# Compute class weights - Handle Class Imbalance
y_train_np = y_train_tensor.numpy()
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
pos_weight = torch.tensor(class_weights[1], dtype=torch.float32).to(device)

# Initialize Model, Function And Optimizer
model_hybrid_bilstm = HybridBiLSTM(vocab_size=len(word2idx), embedding_dim=embed
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = optim.Adam(model_hybrid_bilstm.parameters(), lr=learning_rate)
```

In [244…
```
# Training The Hybrid Bidirectional LSTM Model
for epoch in range(epochs):
    model_hybrid_bilstm.train()
    total_loss = 0
    for texts, structured, labels in train_loader_hybrid:
        texts = texts.to(device)
        structured = structured.to(device)
        labels = labels.to(device).unsqueeze(1)
        optimizer.zero_grad()
        logits = model_hybrid_bilstm(texts, structured)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader_hybrid)
    print(f"Epoch [{epoch+1}/{epochs}] - Loss: {avg_loss:.4f}")
```

```
Epoch [1/10] - Loss: 0.6046
Epoch [2/10] - Loss: 0.4969
Epoch [3/10] - Loss: 0.4505
Epoch [4/10] - Loss: 0.3928
Epoch [5/10] - Loss: 0.3261
Epoch [6/10] - Loss: 0.2794
Epoch [7/10] - Loss: 0.1790
Epoch [8/10] - Loss: 0.1079
Epoch [9/10] - Loss: 0.0702
Epoch [10/10] - Loss: 0.0463
```

In [245…
```
# Evaluating The Hybrid Bidirectional LSTM Model
def evaluate_hybrid_bilstm(model, val_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in val_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy().flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())
    f1 = f1_score(all_labels, all_preds)
    return round(f1, 4)

print("\nHybrid Bidirectional LSTM Validation Performance")
print("--------------------------------------------------")
score = evaluate_hybrid_bilstm(model_hybrid_bilstm, val_loader_hybrid)
print("F1-score:", score)
```

```
Hybrid Bidirectional LSTM Validation Performance
--------------------------------------------------
F1-score: 0.7365
```

The **Hybrid Bidirectional LSTM** achieved an F1-score of **0.7365** on the validation set, which is lower than the F1-scores of the Hybrid CNN, Hybrid RNN, and Hybrid LSTM models. This result may seem surprising because in theory, a bidirectional architecture should provide a richer representation by capturing context from both past and future tokens, which is often expected to improve performance.

Several factors could explain this lower-than-expected performance:

1. **Overfitting**: The hybrid BiLSTM has a large number of parameters due to the bidirectional LSTM combined with structured features. On a relatively small dataset like Dreaddit, this can lead the model to memorize training data patterns instead of generalizing well to unseen validation data.

2. **Optimization Instability**: Bidirectional LSTMs are more complex and require careful tuning of learning rates, hidden dimensions and regularization. Too aggressive optimization or insufficient regularization (eg. dropout) could result in poorer validation accuracy despite low training loss.

3. **Mismatch Between Task and Architecture**: Stress detection in text often relies heavily on local lexical patterns, key phrases and n-grams. CNNs are particularly effective at capturing such local features, whereas recurrent models including BiLSTM focus more on sequential dependencies, which may not be as informative for this dataset.

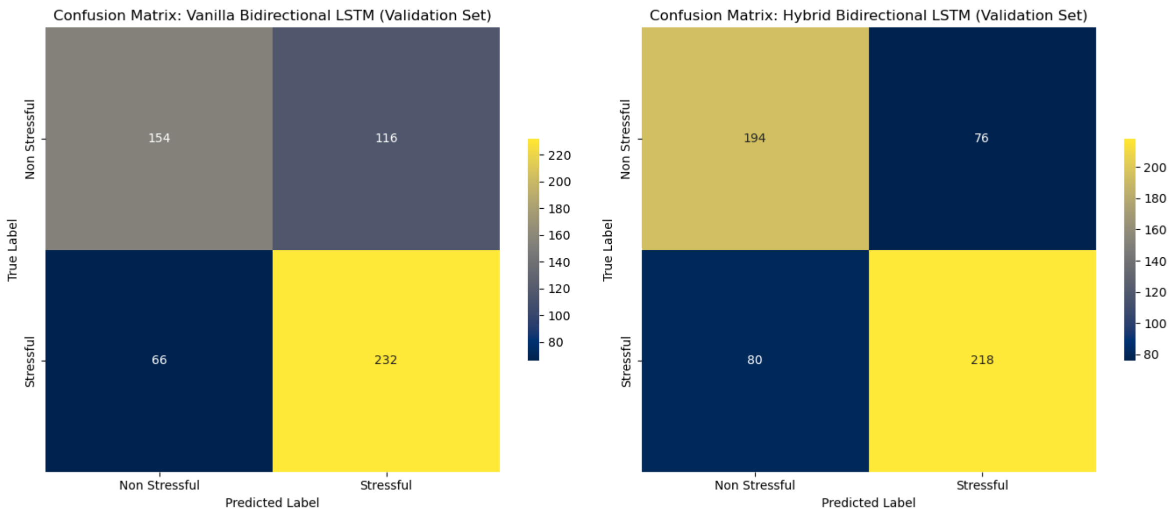
While Hybrid BiLSTM is theoretically powerful, its complexity may have led to overfitting, unstable optimization or inefficiency in capturing the most predictive local patterns in the Dreaddit dataset, resulting in lower validation F1 performance.

Now let us explore the confusion matrix of these two bidirectional LSTM models and explore the differences:

In [247…
```python
# Prediction functions
def get_text_predictions_bilstm(model, data_loader):
    """Predictions for Vanilla BiLSTM (text only)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, labels in data_loader:
            texts = texts.to(device)
            logits = model(texts)  # output logits
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

def get_hybrid_predictions_bilstm(model, data_loader):
    """Predictions for Hybrid BiLSTM (text + structured features)"""
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for texts, structured, labels in data_loader:
            texts = texts.to(device)
            structured = structured.to(device)
            logits = model(texts, structured)
            probs = torch.sigmoid(logits)
            preds = (probs >= 0.5).int().cpu().numpy()
            all_preds.extend(preds.flatten())
            all_labels.extend(labels.numpy())
    return all_labels, all_preds

# Get predictions
y_val_labels_vanilla_bilstm, y_val_pred_vanilla_bilstm = get_text_predictions_bi
y_val_labels_hybrid_bilstm, y_val_pred_hybrid_bilstm = get_hybrid_predictions_bi
```

In [248…

```
# Compute confusion matrices
cm_vanilla_bilstm = confusion_matrix(y_val_labels_vanilla_bilstm, y_val_pred_var
cm_hybrid_bilstm = confusion_matrix(y_val_labels_hybrid_bilstm, y_val_pred_hybri

cms = [cm_vanilla_bilstm, cm_hybrid_bilstm]
titles = ['Vanilla Bidirectional LSTM', 'Hybrid Bidirectional LSTM']
class_names = ['Non Stressful', 'Stressful']

# Plot confusion matrices
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
for ax, cm, title in zip(axes, cms, titles):
    sns.heatmap(cm, annot=True, fmt='d', ax=ax, cmap='cividis',
                xticklabels=class_names, yticklabels=class_names,
                cbar=True, cbar_kws={'shrink':0.5})
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('True Label')
    ax.set_title(f'Confusion Matrix: {title} (Validation Set)')

plt.tight_layout()
plt.show()
```



Key Observations:

Vanilla Bidirectional LSTM:

- Correct predictions: 154 Non-Stressful, 232 Stressful
- Misclassifications: 116 Non-Stressful, 66 Stressful
- Performs reasonably well but shows a tendency to misclassify Non-Stressful posts as Stressful

Hybrid Bidirectional LSTM:

- Correct predictions: 194 Non-Stressful, 218 Stressful
- Misclassifications: 76 Non-Stressful, 80 Stressful
- Improves Non-Stressful detection compared to Vanilla, though slightly worse on Stressful detection

Overall Accuracy:

Hybrid Bidirectional LSTM achieves higher accuracy overall. Hybrid Bidirectional LSTM reduces false alarms significantly. Hybrid Bidirectional LSTM sacrifices a bit of sensitivity to Stressful posts, missing slightly more than Vanilla Bidirectional LSTM. Vanilla Bidirectional LSTM leans toward better Stressful detection. Hybrid Bidirectional LSTM balances both classes better, especially improving Non-Stressful recognition.

Possible Insights:

Vanilla Bidirectional LSTM is stronger at detecting Stressful posts, but at the cost of more false alarms. Hybrid Bidirectional LSTM improves balance, reducing false positives and achieving higher overall accuracy, though it misses a few more Stressful posts. If the goal is to minimize false alarms (important in real-world monitoring where over-alerting can cause fatigue), Hybrid Bidirectional LSTM is preferable. If the priority is to catch as many Stressful posts as possible, Vanilla Bidirectional LSTM may still be competitive despite lower overall accuracy.

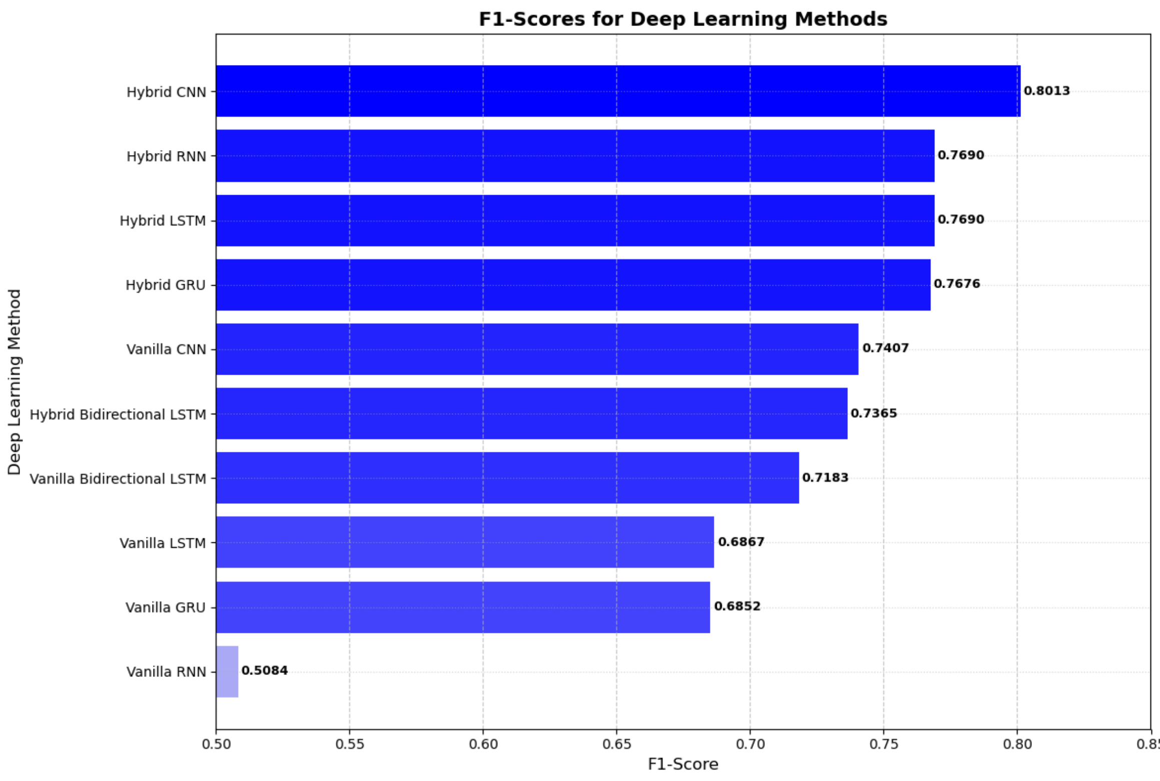
Visualizing The Accuracy Of The Deep Learning Models:

We can now plot the F1-Scores of all the deep learning methods that we have used in a bar chart to compare them side-by-side. We wil also give some key observations and possible insights to the graphs. The deep learning models include:

- **Vanilla CNN**
- **Hybrid CNN**
- **Vanilla RNN**
- **Hybrid RNN**
- **Vanilla LSTM**
- **Hybrid LSTM**
- **Vanilla GRU**
- **Hybrid GRU**
- **Vanilla Bidirectional LSTM**
- **Hybrid Bidirectional LSTM**

In [251…
```
# Prepare the data
methods = ["Vanilla CNN", "Hybrid CNN", "Vanilla RNN", "Hybrid RNN", "Vanilla LS
           "Hybrid LSTM", "Vanilla GRU", "Hybrid GRU", "Vanilla Bidirectional LS
           "Hybrid Bidirectional LSTM"]

f1_scores = [0.7407, 0.8013, 0.5084, 0.7690, 0.6867, 0.7690, 0.6852, 0.7676, 0.7

# Sort by F1-score descending
data_sorted = sorted(zip(methods, f1_scores), key=lambda x: x[1], reverse=True)
```

```
methods_sorted, f1_scores_sorted = zip(*data_sorted)

# Normalize F1-scores to 0-1 for color intensity
min_score = min(f1_scores_sorted)
max_score = max(f1_scores_sorted)
normalized_scores = [(score - min_score) / (max_score - min_score) for score in

# Create colors: higher F1 → darker blue
colors = [sns.light_palette("blue", as_cmap=True)(0.3 + 0.7*norm) for norm in nc

# Plot
plt.figure(figsize=(12, 8))
bars = plt.barh(methods_sorted, f1_scores_sorted, color=colors)

# Add numbers to the right
for bar, score in zip(bars, f1_scores_sorted):
    plt.text(bar.get_width() + 0.001, bar.get_y() + bar.get_height()/2,
             f"{score:.4f}", va='center', ha='left', fontsize=9, fontweight='bol

# Styling
plt.xlabel("F1-Score", fontsize=12)
plt.ylabel("Deep Learning Method", fontsize=12)
plt.title("F1-Scores for Deep Learning Methods", fontsize=14, fontweight='bold')
plt.xlim(0.5, 0.85)
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.grid(axis='y', linestyle=':', alpha=0.5)
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()
```



Key Observations:

1. **Hybrid Models Outperform Vanilla Models Overall**

   The top four methods are all hybrid, with Hybrid CNN achieving the highest F1-score (0.8013). This confirms that combining textual features with additional structured features (LIWC, sentiment) consistently improves performance over vanilla models.

2. **CNN-Based Architectures Dominate**

   Both Hybrid CNN (0.8013) and Vanilla CNN (0.7407) outperform RNN/LSTM/GRU-based models. CNNs appear particularly effective at capturing local lexical patterns and key n-grams, which are critical for stress classification.

3. **Hybridization Benefits Vary By Architecture**

- CNN: Moderate boost (0.8013 vs 0.7407)
- RNN: Large boost (0.7690 vs 0.5084)
- LSTM: Moderate boost (0.7690 vs 0.6867)
- GRU: Moderate boost (0.7676 vs 0.6852)
- Bidirectional LSTM: Slight boost (0.7365 vs 0.7183)

   Hybridization clearly benefits all recurrent architectures, though the magnitude varies. RNNs show the largest relative improvement, reflecting the power of structured features to complement otherwise simple sequential models.

4. **Vanilla RNN Anomaly**

   The Vanilla RNN performs surprisingly poorly (0.5084) compared to other vanilla models. This is likely due to its limited capacity to capture long-range dependencies and complex textual patterns without the assistance of additional features or gating mechanisms (as in LSTM/GRU).

While the deep learning models we have explored, ranging from CNNs to RNNs, LSTMs, GRUs and Bidirectional LSTMs demonstrate the ability to capture textual and structured patterns, they still exhibit several limitations that constrain their performance on the Dreaddit stress classification task:

- **Limited Contextual Understanding**: Although recurrent models like LSTM, GRU and BiLSTM capture sequential dependencies, they may struggle with long-range dependencies or nuanced contextual relationships in posts. CNNs capture local n-gram patterns well but fail to fully incorporate the broader semantic context of sentences.
- **Data Efficiency**: Many of these models require substantial amounts of labeled data to generalize effectively. With a relatively small or imbalanced dataset, they are prone to overfitting, especially hybrid models that combine text and structured features.
- **Feature Engineering Dependency**: While hybrid models attempt to leverage structured features like LIWC scores, these models still rely heavily on careful

preprocessing and manual feature selection. The models may fail to fully exploit complex linguistic or semantic patterns present in the text.

Now let us proceed to the transformer-based models, such as BERT, DistilBert and RoBERTa

---

Transformer-Based NLP Models

A **transformer** is a deep learning architecture that revolutionized natural language processing by relying entirely on a mechanism called self-attention. Unlike traditional RNNs that process text sequentially, transformers process all tokens in parallel and learn contextual relationships by computing how strongly each word attends to every other word in a sentence. This allows the model to capture long-range dependencies, subtle semantic nuances and contextual meaning more effectively.

Models such as **BERT (Bidirectional Encoder Representations from Transformers)**, **RoBERTa**, and **DistilBERT** are pre-trained transformer models that learn rich language representations from massive corpora and can then be fine-tuned on downstream tasks like text classification.

In the Dreaddit training dataset, transformers are fine-tuned on the Text column to classify Reddit posts as stressful or non-stressful. First, each post is tokenized using the tokenizer specific to the chosen transformer architecture, ensuring consistency with its pre-training vocabulary and subword encoding scheme. Special tokens such as `[CLS]` (classification token) and `[SEP]` are added, and the text is converted into input IDs and attention masks. During fine-tuning, the contextual embedding corresponding to the `[CLS]` token is used as a holistic representation of the entire post. This embedding captures nuanced emotional cues, linguistic patterns and contextual signals that are critical for detecting stress-related language.

Transformers are considered state-of-the-art for this task because stress detection depends heavily on contextual understanding rather than isolated keywords. For example, phrases like “I'm fine” may indicate resilience in one context but sarcasm or distress in another. The bidirectional attention mechanism in models like BERT enables deeper semantic interpretation compared to traditional bag-of-words or even standard LSTM approaches.

Approach For The Transformer Models:

In this project, we experiment with three transformer-based models: **BERT, DistilBERT, and RoBERTa**. To manage computational constraints and avoid excessively long installation and loading times within the Jupyter Notebook environment, only the most optimized versions of these models are executed locally.

To demonstrate that comprehensive hyperparameter tuning was conducted, screenshots of the tuning process and results will be provided in the project’s GitHub repository. The hyperparameter optimization was performed on Google Colab using a T4 GPU, which offers significantly faster training times compared to a standard CPU environment. For instance, training a BERT model for one epoch takes approximately 25 - 30 minutes in Jupyter Notebook (CPU), whereas the same process takes under one minute on Google Colab with GPU acceleration. Due to this substantial efficiency gain, model training and tuning were carried out on Google Colab. However, I have run the same code on Jupyter Notebook. The F1-score between the two platforms differ a bit due to the fact that a GPU is used in Google Colab while a CPU is used in Jupyter Notebook. The F1-scores that are listed in the tables below follow the Jupyter Notebook (CPU Version) F1-scores. Overall, the F1-scores on Google Colab are slightly higher compared to Jupyter Notebook. We will take the F1-scores on Jupyter Notebook for model analysis.

Nevertheless, all final results, evaluations and relevant outputs will be presented within the Jupyter Notebook to ensure completeness and reproducibility of the project.

Transformer 1: BERT

BERT is a transformer-based language model that learns deep contextual representations of text by reading words bidirectionally (left and right context simultaneously). Unlike traditional models that process text sequentially, BERT uses self-attention mechanisms to capture complex semantic relationships, long-range dependencies and subtle linguistic cues within sentences. Because it is pre-trained on massive corpora and then fine-tuned for specific tasks, BERT generally performs exceptionally well on text classification problems.

In the Dreaddit dataset, BERT is applied to identify stressful cases by analysing the raw Reddit post text. The model takes tokenised text as input and generates contextual embeddings that capture indicators of stress such as negative emotions, urgency, uncertainty and personal distress. During fine-tuning, BERT learns patterns associated with the binary label (stress vs non-stress). In the hybrid approach (raw text + numerical features), the textual embedding produced by BERT, typically the pooled output corresponding to the CLS token, is concatenated with structured numerical features (eg. post length, sentiment scores or metadata). These combined features are then passed through additional fully connected layers to produce the final classification. This hybrid strategy allows the model to leverage both deep semantic understanding from text and complementary quantitative signals, potentially improving predictive performance compared to using text alone. **We will be using the hybrid models for all three transformer types.**

For the **hyperparameter tuning**, the following hyperparamaters are tuned.

- `learning_rate` : [1e-5, 2e-5, 3e-5, 5e-5]
- `batch_size` : [8, 16, 32]

Hence, on Google Colab, there will be **12 different combinations of hyperparameter sets**. I have used an epoch of 1 for all hyperparameter combinations to prevent excessive runtime on Jupyter Notebook when selecting the best hyperparameters for the BERT model. For the random seed, it is set to 42 for all hyperparameters.

These are the results:

Model Type	Learning Rate	Batch Size	F1-Score After 1 Epoch
BERT	1e-5	8	0.8244
BERT	1e-5	16	0.8144
BERT	1e-5	32	0.7932
BERT	2e-5	8	0.8131
BERT	2e-5	16	0.7994
BERT	2e-5	32	0.8000
BERT	3e-5	8	0.8129
BERT	3e-5	16	0.7734
BERT	3e-5	32	0.7855
BERT	5e-5	8	0.7733
BERT	5e-5	16	0.8113
BERT	5e-5	32	0.7935

From the various combinations of `learning_rate` and `batch_size`, we observe that the highest F1-score acheived for all 12 combinations of hyperparameters is when `learning_rate` is **1e-5** and `batch_size` is **8**, with an F1-score of **0.8244**. Using these hyperparameters, we will show this in jupyter notebook for reference.

For the other hyperparameter combinations of the Hybrid BERT Model, please refer to the GitHub Link:
https://github.com/tadamaen/DSA4262/tree/main/DSA4262%20Individual%20Assignment%
Scores

In [254…

```
# Reproducibility
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(42)

# Imports
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertModel
from torch.optim import AdamW
from tqdm import tqdm
```

In [255…
```
# Feature Definitions
target_col = 'Label'
numeric_features = ['Negative Emotional Language', 'Anxiety', 'Sadness', 'Anger'
                    'Cognitive Processing', 'Introspective Language', 'Casual Re
                    'Tentative Language', 'Certainty', 'First Person Singular Pr
                    'First Person Plural Pronouns', 'Second Person Pronouns',
                    'Third Person Singular Pronouns', 'Third Person Plural Pronc
                    'Social Interactions', 'Family', 'Friends', 'Work', 'Money',
                    'Achievement', 'Risk', 'Social Confidence', 'Authentic', 'Tc
                    'Emotional Activation', 'Mental Imagery', 'Emotional Valence
                    'Sentiment', 'Upvote Proportion', 'Number Of Comments']
categorical_features = ['Subcategory']
ordinal_features = ['Text Length']
sentence_range_order = [['Very Short Text', 'Short Text', 'Medium Text', 'Long T

# Load Dataset
df = dreaddit_train.copy()
df = df.drop(columns=['Confidence'], errors='ignore')
df = df.dropna()
X_structured = df[numeric_features + categorical_features + ordinal_features]
X_text = df['Text']
y = df[target_col]

# Structured Feature Encoding
preprocessor = ColumnTransformer(
    transformers=[('cat', OneHotEncoder(drop='first', handle_unknown='ignore'),
                  ('ord', OrdinalEncoder(categories=sentence_range_order), ordir
                  ('num', StandardScaler(), numeric_features)])

X_structured_processed = preprocessor.fit_transform(X_structured)
if hasattr(X_structured_processed, "toarray"):
    X_structured_processed = X_structured_processed.toarray()
```

In [256…
```
# Train-Validation Split
train_texts, val_texts, train_struct, val_struct, train_labels, val_labels = tra
    X_text.tolist(), X_structured_processed, y.tolist(), test_size=0.2, random_s

# Tokenization
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
train_encodings = tokenizer(train_texts, truncation=True, padding='max_length',
val_encodings = tokenizer(val_texts, truncation=True, padding='max_length', max_

# Custom Dataset
class DreadditHybridDataset(Dataset):
    def __init__(self, encodings, structured_features, labels):
        self.encodings = encodings
        self.structured_features = structured_features
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items
        item['structured'] = torch.tensor(self.structured_features[idx], dtype=t
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = DreadditHybridDataset(train_encodings, train_struct, train_label
val_dataset = DreadditHybridDataset(val_encodings, val_struct, val_labels)
```

In [257…
```
# Defining Hybrid BERT Model
class HybridBERT(nn.Module):
    def __init__(self, structured_dim):
        super(HybridBERT, self).__init__()
        self.bert = BertModel.from_pretrained('bert-base-uncased')
        self.dropout = nn.Dropout(0.3)
        self.classifier = nn.Linear(self.bert.config.hidden_size + structured_di

    def forward(self, input_ids, attention_mask, structured, labels=None):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        combined = torch.cat((pooled_output, structured), dim=1)
        combined = self.dropout(combined)
        logits = self.classifier(combined)
        loss = None
        if labels is not None:
            loss_fn = nn.CrossEntropyLoss()
            loss = loss_fn(logits, labels)
        return {"loss": loss, "logits": logits}

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

In [258…
```
import logging
from transformers import logging as hf_logging

hf_logging.set_verbosity_error()
logging.getLogger("transformers").setLevel(logging.ERROR)

# Training With Best Hyperparameters
learning_rate = 1e-5
batch_size = 8
epochs = 1

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
model = HybridBERT(structured_dim=train_struct.shape[1])
model.to(device)
optimizer = AdamW(model.parameters(), lr=learning_rate)
```

```
Loading weights:   0%|          | 0/199 [00:00<?, ?it/s]
```

In [259…
```
# Training
model.train()

for epoch in range(epochs):
    total_loss = 0
    for batch in tqdm(train_loader):
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        structured = batch['structured'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, stru
        loss = outputs["loss"]
        total_loss += loss.item()
        loss.backward()
        optimizer.step()
    print(f"Training Loss: {total_loss/len(train_loader)}")
```

```
# Validation
model.eval()
predictions, true_labels = [], []

with torch.no_grad():
    for batch in val_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        structured = batch['structured'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, stru
        logits = outputs["logits"]
        preds = torch.argmax(logits, dim=1)
        predictions.extend(preds.cpu().numpy())
        true_labels.extend(labels.cpu().numpy())

f1 = f1_score(true_labels, predictions)

print()
print(f"Learning Rate: {learning_rate}, Batch Size: {batch_size}")
print()
print("Hybrid BERT Validation Performance")
print("----------------------------------")
print(f"F1 Score: {round(f1, 4)}")
```

```
100%|██████████| 284/284 [21:38<00:00,  4.57s/it]
Training Loss: 0.5891403710128555

Learning Rate: 1e-05, Batch Size: 8

Hybrid BERT Validation Performance
----------------------------------
F1 Score: 0.8244
```

Key Observations:

The F1-scores achieved with the BERT-based transformer models consistently **range between 0.77 and 0.83**, with an average performance of approximately **0.80 to 0.81**. These results surpass all classical machine learning and deep learning baselines explored, including the previously strongest hybrid CNN model. Notably, certain hyperparameter configurations yield performance levels that exceed the best deep learning benchmark. Among the twelve tested combinations, the highest F1-score recorded was **0.8244**, achieved with a **learning rate of 1e-5 and a batch size of 8**.

Key Insights:

Transformer-based architectures demonstrate clear superiority over traditional ML and CNN-based approaches for this task. Fine-tuning hyperparameters, particularly learning rate and batch size, has a significant impact on performance, generally with smaller batch sizes paired with lower learning rates yielding optimal results. The consistently strong performance of BERT highlights the value of leveraging pre-trained language models for complex text classification problems, especially when combined with structured features.

Now we will explore another type of BERT model - **DistilBERT**.

Transformer 2: DistilBERT

DistilBERT is a lighter, faster and more efficient version of the original BERT model. It is created through a process called knowledge distillation, where the large BERT model (the teacher) transfers its knowledge to a smaller model (the student). While traditional BERT is powerful, it is computationally heavy and resource-intensive, making it slower to train and deploy. DistilBERT retains about 95% of BERT's language understanding capabilities but is approximately 40% smaller and runs about 60% faster.

For the dreaddit dataset, which involves classifying text into stressful and non-stressful categories, DistilBERT is a strong choice because it balances accuracy with efficiency. The dataset requires nuanced understanding of emotional and psychological language, and DistilBERT's pre-trained contextual embeddings capture these subtleties effectively. At the same time, its reduced size and faster runtimes make it more practical for experimentation, hyperparameter tuning, and deployment in real-world applications where computational resources may be limited.

Similarly, for the **hyperparameter tuning**, the following hyperparamaters are tuned.

- `learning_rate` : [1e-5, 2e-5, 3e-5, 5e-5]
- `batch_size` : [8, 16, 32]

Hence, on Google Colab, there will be **12 different combinations of hyperparameter sets**. I have used an epoch of 1 for all hyperparameter combinations to prevent excessive runtime on Jupyter Notebook when selecting the best hyperparameters for the DistilBERT model. For the random seed, it is set to 42 for all hyperparameters.

These are the results:

Model Type	Learning Rate	Batch Size	F1-Score After 1 Epoch
DistilBERT	1e-5	8	0.8251
DistilBERT	1e-5	16	0.8000
DistilBERT	1e-5	32	0.8019
DistilBERT	2e-5	8	0.8228
DistilBERT	2e-5	16	0.8011
DistilBERT	2e-5	32	0.7885
DistilBERT	3e-5	8	0.8170
DistilBERT	3e-5	16	0.8191
DistilBERT	3e-5	32	0.7896
DistilBERT	5e-5	8	0.8093
DistilBERT	5e-5	16	0.8239
DistilBERT	5e-5	32	0.7935

From the various combinations of `learning_rate` and `batch_size`, we observe that the highest F1-score acheived for all 12 combinations of hyperparameters is when `learning_rate` is **1e-5** and `batch_size` is **8**, with an F1-score of **0.8251**. Using these hyperparameters, we will show this in jupyter notebook for reference.

For the other hyperparameter combinations of the Hybrid DistilBERT Model, please refer to the GitHub Link:
https://github.com/tadamaen/DSA4262/tree/main/DSA4262%20Individual%20Assignment%
Scores

In [262…
```
# Reproducibility
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(42)

# Imports
from transformers import DistilBertTokenizer, DistilBertModel
```

In [263…
```
# Feature Definitions
target_col = 'Label'
numeric_features = ['Negative Emotional Language', 'Anxiety', 'Sadness', 'Anger'
                    'Cognitive Processing', 'Introspective Language', 'Casual Re
                    'Tentative Language', 'Certainty', 'First Person Singular Pr
                    'First Person Plural Pronouns', 'Second Person Pronouns',
                    'Third Person Singular Pronouns', 'Third Person Plural Pronc
                    'Social Interactions', 'Family', 'Friends', 'Work', 'Money',
                    'Achievement', 'Risk', 'Social Confidence', 'Authentic', 'To
                    'Emotional Activation', 'Mental Imagery', 'Emotional Valence
                    'Sentiment', 'Upvote Proportion', 'Number Of Comments']
categorical_features = ['Subcategory']
ordinal_features = ['Text Length']
sentence_range_order = [['Very Short Text', 'Short Text', 'Medium Text', 'Long T

# Load Dataset
df = dreaddit_train.copy()
df = df.drop(columns=['Confidence'], errors='ignore')
df = df.dropna()
X_structured = df[numeric_features + categorical_features + ordinal_features]
X_text = df['Text']
y = df[target_col]

# Structured Feature Encoding
preprocessor = ColumnTransformer(
    transformers=[('cat', OneHotEncoder(drop='first', handle_unknown='ignore'),
                  ('ord', OrdinalEncoder(categories=sentence_range_order), ordir
                  ('num', StandardScaler(), numeric_features)])
X_structured_processed = preprocessor.fit_transform(X_structured)
```

```
if hasattr(X_structured_processed, "toarray"):
    X_structured_processed = X_structured_processed.toarray()
```

In [264…
```
# Train-Validation Split
train_texts, val_texts, train_struct, val_struct, train_labels, val_labels = tra
    X_text.tolist(), X_structured_processed, y.tolist(), test_size=0.2, random_s

# Tokenization
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
train_encodings = tokenizer(train_texts, truncation=True, padding='max_length', 
val_encodings = tokenizer(val_texts, truncation=True, padding='max_length', max_

# Custom Dataset
class DreadditHybridDataset(Dataset):
    def __init__(self, encodings, structured_features, labels):
        self.encodings = encodings
        self.structured_features = structured_features
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items
        item['structured'] = torch.tensor(self.structured_features[idx], dtype=t
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = DreadditHybridDataset(train_encodings, train_struct, train_label
val_dataset = DreadditHybridDataset(val_encodings, val_struct, val_labels)
```

In [265…
```
# Defining Hybrid DistilBERT Model
class HybridDistilBERT(nn.Module):
    def __init__(self, structured_dim):
        super(HybridDistilBERT, self).__init__()
        self.bert = DistilBertModel.from_pretrained('distilbert-base-uncased')
        self.dropout = nn.Dropout(0.3)
        self.classifier = nn.Linear(self.bert.config.hidden_size + structured_di

    def forward(self, input_ids, attention_mask, structured, labels=None):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs.last_hidden_state[:,0]  # DistilBERT does not ha
        combined = torch.cat((pooled_output, structured), dim=1)
        combined = self.dropout(combined)
        logits = self.classifier(combined)
        loss = None
        if labels is not None:
            loss_fn = nn.CrossEntropyLoss()
            loss = loss_fn(logits, labels)
        return {"loss": loss, "logits": logits}

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

In [266…
```
# Training With Best Hyperparameters
import logging
from transformers import logging as hf_logging
hf_logging.set_verbosity_error()
logging.getLogger("transformers").setLevel(logging.ERROR)
```

```
learning_rate = 1e-5
batch_size = 8
epochs = 1

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
model = HybridDistilBERT(structured_dim=train_struct.shape[1])
model.to(device)
optimizer = AdamW(model.parameters(), lr=learning_rate)
```

```
Loading weights:   0%|          | 0/100 [00:00<?, ?it/s]
```

In [267…

```
# Training
model.train()

for epoch in range(epochs):
    total_loss = 0
    for batch in tqdm(train_loader):
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        structured = batch['structured'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, stru
        loss = outputs["loss"]
        total_loss += loss.item()
        loss.backward()
        optimizer.step()
    print(f"Training Loss: {total_loss/len(train_loader)}")

# Validation
model.eval()
predictions, true_labels = [], []

with torch.no_grad():
    for batch in val_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        structured = batch['structured'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, stru
        logits = outputs["logits"]
        preds = torch.argmax(logits, dim=1)
        predictions.extend(preds.cpu().numpy())
        true_labels.extend(labels.cpu().numpy())

f1 = f1_score(true_labels, predictions)

print()
print(f"Learning Rate: {learning_rate}, Batch Size: {batch_size}")
print()
print("Hybrid DistilBERT Validation Performance")
print("----------------------------------------")
print(f"F1 Score: {round(f1, 4)}")
```

```
100%|██████████| 284/284 [10:17<00:00,  2.17s/it]
```

```
Training Loss: 0.5139296406119223

Learning Rate: 1e-05, Batch Size: 8

Hybrid DistilBERT Validation Performance
----------------------------------------
F1 Score: 0.8251
```

Key Observations:

The DistilBERT model demonstrates consistently strong performance, achieving F1-scores in the **range of 0.78 to 0.83**, closely aligned with the results from the full BERT model. Importantly, DistilBERT shows a slight edge: a **greater number of hyperparameter configurations surpass the 0.80 threshold**, with **several exceeding 0.82**. Even the lowest-performing configuration yields an F1-score of 0.7885, which is notably higher than BERT's lowest score of 0.7733. This indicates that DistilBERT not only matches but marginally improves upon BERT's effectiveness in this task. The best performing DistilBERT model has an F1-score of **0.8251**, achieved with a **learning rate of 1e-5 and a batch size of 8**.

Key Insights:

DistilBERT achieves comparable or better performance than BERT while being faster and more resource-efficient, making it highly practical for large-scale or real-time applications. The model's ability to consistently deliver strong results across different learning rates and batch sizes suggests greater stability and reliability compared to BERT. Given its smaller size and faster runtimes, DistilBERT is particularly well-suited for scenarios where computational resources are limited but high accuracy like F1-score is still required. The model's superior balance of speed and accuracy makes it an excellent candidate for applications like the dreaddit dataset, where nuanced language understanding is critical but efficiency is equally important.

Now let us proceed to look at the third and final transformer that we will be considering, **RoBERTa**.

Transformer 3: RoBERTa

RoBERTa (Robustly Optimized BERT Approach) is an advanced variant of BERT that improves upon the original model by optimizing its training methodology. Unlike traditional BERT, which was trained with certain constraints such as a fixed number of training steps and the use of next-sentence prediction, RoBERTa removes the next-sentence prediction objective, trains on much larger datasets and uses dynamic masking strategies. Compared to DistilBERT, which is a compressed and faster version of BERT, RoBERTa is designed to maximize accuracy and robustness, often outperforming both BERT and DistilBERT in benchmark tasks due to its more extensive pretraining and refined optimization

For the dreaddit dataset, which involves distinguishing between stressful and non-stressful text, RoBERTa is particularly well-suited because it captures subtle linguistic cues and contextual nuances more effectively than earlier transformer models. Stress-related language often relies on fine-grained emotional and psychological signals, and RoBERTa's deeper training allows it to model these complexities with greater precision. While it is more computationally demanding than DistilBERT, the trade-off is higher accuracy and stronger generalization, making RoBERTa a powerful choice for tasks requiring nuanced text classification such as stress detection.

Similarly, for the **hyperparameter tuning**, the following hyperparamaters are tuned.

- `learning_rate` : [1e-5, 2e-5, 3e-5, 5e-5]
- `batch_size` : [8, 16, 32]

Hence, on Google Colab, there will be **12 different combinations of hyperparameter sets**. I have used an epoch of 1 for all hyperparameter combinations to prevent excessive runtime on Jupyter Notebook when selecting the best hyperparameters for the RoBERTa model. For the random seed, it is set to 42 for all hyperparameters.

These are the results:

Model Type	Learning Rate	Batch Size	F1-Score After 1 Epoch
RoBERTa	1e-5	8	0.8033
RoBERTa	1e-5	16	0.8150
RoBERTa	1e-5	32	0.8059
RoBERTa	2e-5	8	0.8096
RoBERTa	2e-5	16	0.8287
RoBERTa	2e-5	32	0.7723
RoBERTa	3e-5	8	0.7967
RoBERTa	3e-5	16	0.8141
RoBERTa	3e-5	32	0.8242
RoBERTa	5e-5	8	0.7820
RoBERTa	5e-5	16	0.8195
RoBERTa	5e-5	32	0.8207

From the various combinations of `learning_rate` and `batch_size`, we observe that the highest F1-score acheived for all 12 combinations of hyperparameters is when `learning_rate` is **2e-5** and `batch_size` is **16**, with an F1-score of **0.8287**. Using these hyperparameters, we will show this in jupyter notebook for reference.

For the other hyperparameter combinations of the Hybrid RoBERTa Model, please refer to the GitHub Link:
https://github.com/tadamaen/DSA4262/tree/main/DSA4262%20Individual%20Assignment%
Scores

In [270…
```
# Reproducibility
def set_seed(seed=42):
    import random, numpy as np, torch
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(42)

# Imports
from transformers import RobertaTokenizer, RobertaModel
```

In [271…
```
# Feature Definitions
target_col = 'Label'
numeric_features = ['Negative Emotional Language', 'Anxiety', 'Sadness', 'Anger'
                    'Cognitive Processing', 'Introspective Language', 'Casual Re
                    'Tentative Language', 'Certainty', 'First Person Singular Pr
                    'First Person Plural Pronouns', 'Second Person Pronouns',
                    'Third Person Singular Pronouns', 'Third Person Plural Pronc
                    'Social Interactions', 'Family', 'Friends', 'Work', 'Money',
                    'Achievement', 'Risk', 'Social Confidence', 'Authentic', 'Tc
                    'Emotional Activation', 'Mental Imagery', 'Emotional Valence
                    'Sentiment', 'Upvote Proportion', 'Number Of Comments']
categorical_features = ['Subcategory']
ordinal_features = ['Text Length']
sentence_range_order = [['Very Short Text', 'Short Text', 'Medium Text', 'Long T

# Load Dataset
df = dreaddit_train.copy()
df = df.drop(columns=['Confidence'], errors='ignore')
df = df.dropna()
X_structured = df[numeric_features + categorical_features + ordinal_features]
X_text = df['Text']
y = df[target_col]

# Structured Feature Encoding
preprocessor = ColumnTransformer(
    transformers=[('cat', OneHotEncoder(drop='first', handle_unknown='ignore'),
                  ('ord', OrdinalEncoder(categories=sentence_range_order), ordir
                  ('num', StandardScaler(), numeric_features)])
X_structured_processed = preprocessor.fit_transform(X_structured)
if hasattr(X_structured_processed, "toarray"):
    X_structured_processed = X_structured_processed.toarray()
```

In [272…
```
# Train-Validation Split
train_texts, val_texts, train_struct, val_struct, train_labels, val_labels = tra
    X_text.tolist(), X_structured_processed, y.tolist(), test_size=0.2, random_s

# Tokenization
tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
train_encodings = tokenizer(train_texts, truncation=True, padding='max_length',
val_encodings = tokenizer(val_texts, truncation=True, padding='max_length', max_

# Custom Dataset
```

```
class DreadditHybridDataset(Dataset):
    def __init__(self, encodings, structured_features, labels):
        self.encodings = encodings
        self.structured_features = structured_features
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items
        item['structured'] = torch.tensor(self.structured_features[idx], dtype=t
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = DreadditHybridDataset(train_encodings, train_struct, train_label
val_dataset = DreadditHybridDataset(val_encodings, val_struct, val_labels)
```

In [273…

```
# Defining Hybrid RoBERTa Model
class HybridRoBERTa(nn.Module):
    def __init__(self, structured_dim):
        super(HybridRoBERTa, self).__init__()
        self.roberta = RobertaModel.from_pretrained('roberta-base')
        self.dropout = nn.Dropout(0.3)
        self.classifier = nn.Linear(self.roberta.config.hidden_size + structured

    def forward(self, input_ids, attention_mask, structured, labels=None):
        outputs = self.roberta(input_ids=input_ids, attention_mask=attention_mas
        pooled_output = outputs.last_hidden_state[:,0]  # CLS token representati
        combined = torch.cat((pooled_output, structured), dim=1)
        combined = self.dropout(combined)
        logits = self.classifier(combined)
        loss = None
        if labels is not None:
            loss_fn = nn.CrossEntropyLoss()
            loss = loss_fn(logits, labels)
        return {"loss": loss, "logits": logits}

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

In [274…

```
# Training With Best Hyperparameters
import logging
from transformers import logging as hf_logging
hf_logging.set_verbosity_error()
logging.getLogger("transformers").setLevel(logging.ERROR)

learning_rate = 2e-5
batch_size = 16
epochs = 1

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
model = HybridRoBERTa(structured_dim=train_struct.shape[1])
model.to(device)
optimizer = AdamW(model.parameters(), lr=learning_rate)
```

```
Loading weights:   0%|          | 0/197 [00:00<?, ?it/s]
```

In [275…

```
# Training
```

```python
model.train()

for epoch in range(epochs):
    total_loss = 0
    for batch in tqdm(train_loader):
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        structured = batch['structured'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, stru
        loss = outputs["loss"]
        total_loss += loss.item()
        loss.backward()
        optimizer.step()
    print(f"Training Loss: {total_loss/len(train_loader)}")

# Validation
model.eval()
predictions, true_labels = [], []

with torch.no_grad():
    for batch in val_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        structured = batch['structured'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, stru
        logits = outputs["logits"]
        preds = torch.argmax(logits, dim=1)
        predictions.extend(preds.cpu().numpy())
        true_labels.extend(labels.cpu().numpy())

f1 = f1_score(true_labels, predictions)

print()
print(f"Learning Rate: {learning_rate}, Batch Size: {batch_size}")
print()
print("Hybrid RoBERTa Validation Performance")
print("-------------------------------------")
print(f"F1 Score: {round(f1, 4)}")
```

```
100%|██████████| 142/142 [19:55<00:00,  8.42s/it]
Training Loss: 0.5210116394388844

Learning Rate: 2e-05, Batch Size: 16

Hybrid RoBERTa Validation Performance
-------------------------------------
F1 Score: 0.8287
```

Key Observations:

The F1-scores of the RoBERTa models generally fall within **the range of 0.77 to 0.83**, which is comparable to the performance of the BERT and DistilBERT models. Notably, the best-performing RoBERTa model achieved an F1-score of **0.8287**, slightly higher than both the top BERT model (0.8244) and DistilBERT model (0.8251). This happens when learning rate is **2e-5** and batch size is **16**. Across most hyperparameter combinations, RoBERTa consistently maintains F1-scores above 0.80, with several configurations surpassing 0.82, indicating stable and high predictive performance.

These results suggest that while the three transformer-based architectures perform similarly on average, RoBERTa offers a marginal performance advantage when properly tuned. This may be attributed to RoBERTa's improved pretraining strategies, including more extensive training data and longer sequences, which could enhance its ability to capture nuanced textual patterns. DistilBERT, although smaller and more computationally efficient, still achieves comparable performance in many cases, demonstrating that model size reduction does not necessarily lead to significant loss in accuracy.

Comparison Of The 28 Models (Cost-Benefit Analysis):

We will now conduct a comprehensive side-by-side comparison of all 28 models implemented in this study, encompassing classical machine learning approaches, deep learning architectures and transfomer models. The models will be evaluated across three primary dimensions: **predictive performance (F1-score)**, **computational efficiency (run time)**, and **implementation cost (resource requirements)**. This structured cost–benefit analysis is essential to determine the most suitable model for deployment on the `dreaddit_test` dataset, where final performance will be assessed.

For the purposes of this academic project, primary emphasis is placed on the F1-score as it reflects the model's ability to accurately classify stressful and non-stressful texts. Although runtime and implementation cost are less critical in this context given that the models are not intended for immediate industry deployment, they remain relevant considerations for completeness and methodological rigor. Consequently, both runtime and computational cost have been included as comparative dimensions, providing a balanced and comprehensive evaluation of all 28 models prior to selecting the most suitable candidate for final testing.

Model Used	**F1-Score**	**Efficiency (Run Time)**	**Cost Of Model**
Logistic Regression (No Tuning)	Medium - 0.7674	High - Simple linear model, very fast on CPU	Low - Minimal computational resources
Logistic Regression (Grid Search CV)	High - 0.7774	Medium - Grid search over hyperparameters increases runtime	Low - Still linear model, negligible resource increase
Logistic Regression (Random Search CV)	High - 0.7740	High - Randomized search faster than grid search	Low - Still linear model, minimal resource usage
XG Boost (No Tuning)	High - 0.7729	Medium - Boosted trees, scales moderately with dataset size	Medium - Tree-based model uses moderate memory

Model Used	F1-Score	Efficiency (Run Time)	Cost Of Model
XG Boost (Grid Search CV)	Medium - 0.7608	Low - Hyperparameter search is expensive	High - Multiple trees trained repeatedly increase compute and memory usage
XG Boost (Random Search CV)	Medium - 0.7472	Medium - Random search faster than grid search	Medium - Similar to above but fewer models trained
Cat Boost (No Tuning)	Medium - 0.7667	Medium - Tree boosting with categorical handling, moderate runtime	Medium - Requires GPU/CPU memory for tree ensembles
Cat Boost (Grid Search CV)	Medium - 0.7534	Low - Grid search over hyperparameters, slow	High - Training multiple parameter combinations increases cost
Cat Boost (Random Search CV)	Medium - 0.7511	Low - Random search reduces runtime but still slow in CatBoost Models	Medium - Moderate GPU/CPU usage
Light GBM (No Tuning)	Medium - 0.7538	High - LightGBM is optimized, very fast even for large datasets	Low - Memory-efficient boosting implementation
Light GBM (Grid Search CV)	Medium - 0.7629	Medium - Hyperparameter search increases runtime	Medium - Repeated model training increases resource usage
Light GBM (Random Search CV)	Medium - 0.7483	High - Faster than grid search	Medium - Moderate memory usage
SVM (No Tuning)	High - 0.7739	Low - Kernel SVM is slow for large datasets	Medium - Quadratic memory/time complexity for kernel computations
SVM (Grid Search CV)	High - 0.7763	Low - Grid search further increases runtime	High - Multiple kernel evaluations, high computational cost
SVM (Random Search CV)	High - 0.7862	Low - Slightly faster than grid search but still complex	High - Multiple kernel evaluations, high computational cost
Vanilla CNN	Medium - 0.7407	Medium - Single CNN on text, moderate computation, can run on GPU	Medium - Embedding + convolution layers require moderate GPU memory

Model Used	F1-Score	Efficiency (Run Time)	Cost Of Model
Hybrid CNN	High - 0.8013	Medium - Combines text + structured features, slightly more computation than vanilla CNN	Medium - Additional FC layers for structured features slightly increase memory
Vanilla RNN	Extremely Low - 0.5084	Medium - Simple RNN, slower than CNN for long sequences due to sequential processing	Low - Minimal model parameters, low GPU requirement
Hybrid RNN	Medium - 0.7690	Low - Text + structured features increase computation	Medium - More parameters than vanilla RNN but still manageable
Vanilla LSTM	Low - 0.6867	Medium - LSTM processes sequences sequentially, moderate runtime	Medium - More parameters than RNN, requires more GPU memory
Hybrid LSTM	Medium - 0.7690	Medium - STM + structured features, higher memory, slightly longer training	High - Increased computation and GPU memory for structured FC layers
Vanilla GRU	Low - 0.6852	Medium - GRU is faster than LSTM but still sequential	Medium - Similar cost to LSTM but fewer parameters
Hybrid GRU	Medium - 0.7676	Medium - Text + structured features, slightly slower	High - Extra FC layers for structured input increases compute
Vanilla Bidirectional LSTM	Medium - 0.7183	Low - Bidirectional, double the sequence computation	High - Double parameters vs LSTM, higher GPU usage
Hybrid Bidirectional LSTM	Medium - 0.7365	Low - Bidirectional + structured features, highest sequential computation	High - Highest memory and compute requirements due to bidirectionality
Hybrid BERT	Very High - 0.8244	Medium - Transformer-based with large number of parameters, parameters chosen speed up process	High - Requires substantial GPU memory and compute for training and inference
Hybrid DistilBERT	Very High - 0.8251	High - Smaller distilled version of BERT, faster training and inference than full BERT	Medium - Reduced GPU and memory requirements compared to full BERT

Model Used	F1-Score	Efficiency (Run Time)	Cost Of Model
Hybrid RoBERTa	Very High - 0.8287	Medium - Larger and more parameter-heavy than DistilBERT, parameters chosen speed up process	High - Very large model with high computational and memory demands

Analysis of the F1-scores across the 28 models indicates that **transformer-based architectures, including Hybrid BERT, Hybrid DistilBERT, and Hybrid RoBERTa, consistently achieve scores in the range of 0.82 to 0.83**. This performance is **notably higher than that of classical machine learning and conventional deep learning models, most of which achieve F1-scores between 0.75 and 0.78**. This represents an approximate **6–8% improvement**, which is particularly meaningful in the context of healthcare applications. For instance, in a dataset comprising one thousand text cases, a 7% increase in F1-score corresponds to correctly identifying 70 additional cases. This improvement has the potential to enhance patient outcomes and contribute positively to societal well-being by enabling timely intervention for a greater number of individuals.

Assuming a threshold F1-score of **0.8 for model consideration**, **only four models meet this criterion: Hybrid BERT, Hybrid DistilBERT, Hybrid RoBERTa, and the Hybrid CNN**. While all four models demonstrate strong predictive performance, **transformer-based models consistently outperform the Hybrid CNN**, which achieves a slightly lower F1-score of 0.8013. Among the transformer models, **Hybrid RoBERTa achieves the highest F1-score of 0.8287**, marginally surpassing both Hybrid BERT (0.8244) and Hybrid DistilBERT (0.8251). In addition to its superior predictive accuracy, Hybrid RoBERTa benefits from robust pretraining on large-scale text corpora, enhancing its ability to capture complex linguistic patterns and subtle contextual information.

Considering both the performance metrics and its ability to generalize effectively to textual data, Hybrid RoBERTa is therefore selected as the optimal model. For final testing, we will adopt **Hybrid RoBERTa** as the preferred model for evaluating the `dreaddit_test` dataset.

NOTE: For the purpose of this project, where F1-scores have biggest weightage for consideration, Hybrid RoBERTa is chosen. In the real-world industry setting, where cost and runtime of models are as equally/even more important than F1-scores, other models such as Hybrid CNN and Hybrid DistilBERT might be chosen. In budget conscious industries where cost is most important, we need to focus on cheaper yet pretty accurate models such as Logistic Regression with RandomSearchCV.

Implementing Hybrid RoBERTa on Dreaddit Test Dataset:

We will now apply **the Hybrid RoBERTa model** to predict whether texts in the `dreaddit_test` dataset are stressful or non-stressful. Consistent with the validation procedure, we will use the same model architecture and fine-tuning configurations to ensure a fair comparison. To maximize model performance, we will leverage on the initial optimal hypermeters - `learning_rate` of **2e-5** and `batch_size` of **16**, with 1 epoch (same as previously for the validation model). Finally, the model's performance will be evaluated using the F1-score, allowing us to compare results against the validation set and assess the model's consistency and generalization on the unseen test data.

In [279…

```
# Reproducibility for Test Set
set_seed(42)

# Load Test Dataset
df_test = dreaddit_test.copy()
df_test = df_test.drop(columns=['Confidence'], errors='ignore')
df_test = df_test.dropna()
X_structured_test = df_test[numeric_features + categorical_features + ordinal_fe
X_text_test = df_test['Text']
y_test = df_test[target_col]

# Structured Feature Encoding
X_structured_test_processed = preprocessor.transform(X_structured_test)
if hasattr(X_structured_test_processed, "toarray"):
    X_structured_test_processed = X_structured_test_processed.toarray()

# Tokenization (RoBERTa)
test_encodings = tokenizer(X_text_test.tolist(), truncation=True, padding='max_l

# Custom Dataset
test_dataset = DreadditHybridDataset(test_encodings, X_structured_test_processed
test_loader = DataLoader(test_dataset, batch_size=batch_size)

# Evaluation on Test Set (RoBERTa)
model.eval()
predictions, true_labels = [], []

with torch.no_grad():
    for batch in test_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        structured = batch['structured'].to(device)
        labels = batch['labels'].to(device)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, stru
        logits = outputs["logits"]
        preds = torch.argmax(logits, dim=1)
        predictions.extend(preds.cpu().numpy())
        true_labels.extend(labels.cpu().numpy())

f1 = f1_score(true_labels, predictions)

print()
print(f"Learning Rate: {learning_rate}, Batch Size: {batch_size}")
print()
print("Hybrid RoBERTa Test Performance")
print("--------------------------------")
print(f"F1 Score: {round(f1, 4)}")
```

```
Learning Rate: 2e-05, Batch Size: 16

Hybrid RoBERTa Test Performance
-------------------------------
F1 Score: 0.8173
```

Final F1-Score On Test Data: 0.8173

The F1-score of the Hybrid RoBERTa model on the dreaddit_test dataset is **0.8173**, indicating very strong predictive performance. Although this is slightly lower than the F1-score observed on the validation set (0.8287), it remains a robust result. An F1-score of 0.8173 suggests that the model **correctly classifies slightly more than 4 out of 5 texts**, demonstrating reliable performance in practical applications.

The modest reduction in F1-score on the test set compared to the validation set can be attributed to several factors:

1. **Distributional Differences**: The `dreaddit_test` dataset may include texts with slightly different language patterns, phrasing or topical content compared to the training and validation data, which can affect RoBERTa's ability to generalize.

2. **Subtle Contextual Variations**: RoBERTa captures nuanced linguistic and contextual cues. If these subtleties differ between the test set and validation set (eg. differences in how stress is expressed), the model may experience a slight decline in predictive accuracy.

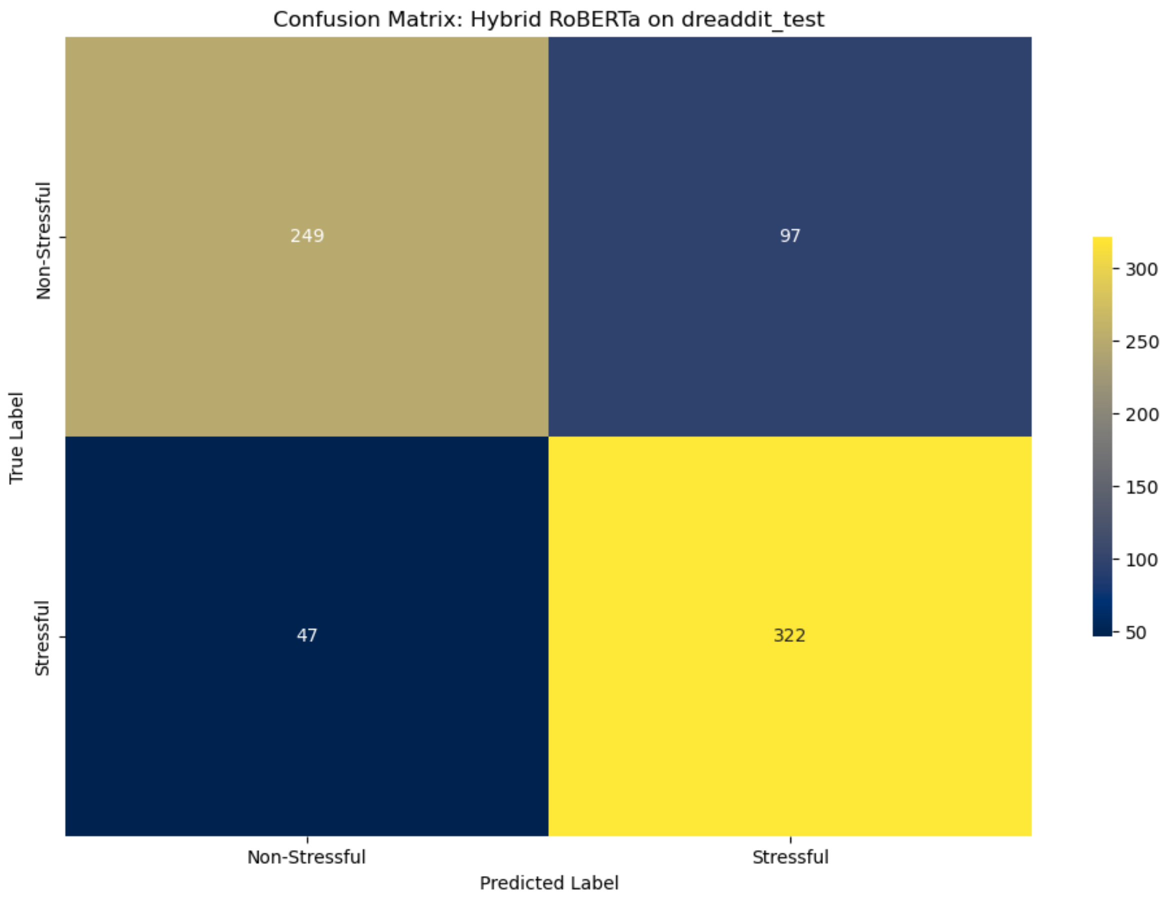
3. **Sample Variance**: The validation set is drawn from the same distribution as the training data, whereas the test set consists of unseen examples, introducing natural variability in performance metrics.

4. **Model Sensitivity to Rare Patterns**: Transformer models like RoBERTa can be sensitive to rare or infrequent textual patterns that may appear in the test set but were underrepresented during training, contributing to slight drops in F1-score.

We will now analyze the confusion matrix to further understand the model's classification behavior and identify which types of errors are most common.

In [281…

```python
# Compute confusion matrix
cm = confusion_matrix(true_labels, predictions)

# Labels for axes
labels = ['Non-Stressful', 'Stressful']

# Plot
plt.figure(figsize=(12, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='cividis',
            xticklabels=labels, yticklabels=labels,
            cbar=True, cbar_kws={'shrink':0.5})
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix: Hybrid RoBERTa on dreaddit_test')
plt.show()
```



Key Observations:

1. **Numbers/Statistical Figures:**

- True Positives (Stressful correctly classified): 322
- True Negatives (Non-Stressful correctly classified): 249
- False Positives (Non-Stressful misclassified as Stressful): 97
- False Negatives (Stressful misclassified as Non-Stressful): 47

2. **Trend/Comparison Of Figures:**

- Significantly more false positives (97) than false negatives (47).
- Correct classifications (571 total) outweigh misclassifications (144 total), suggesting the model has decent predictive power.

Possible Insights:

The model tends to generate more false positives (97) than false negatives (47), meaning that some non-stressful texts are mistakenly flagged as stressful. While this could result in unnecessary alerts, it is generally less critical than missing truly stressful texts. The lower number of false negatives suggests that most stressful texts are correctly identified, which is particularly important in a mental health context where failing to detect stress could lead to missed opportunities for intervention. The ratio of correct to incorrect classifications demonstrates that the model is well-calibrated, but there is still room to improve recall for stressful texts to further reduce the number of missed cases. Overall, the balance between sensitivity and specificity indicates that the Hybrid RoBERTa model is both reliable and practical for classifying stressful versus non-stressful texts, with a slight bias toward caution (flagging some non-stressful texts) rather than risk of omission.

Cost Of False Positives And False Negatives:

False Positives

False positives occur when non-stressful texts are incorrectly classified as stressful. In this context, it means flagging messages that are harmless as potentially concerning. While this can lead to unnecessary alerts, wasted resources or causing worry for someone who is fine, the severity is relatively lower compared to missing actual distress. In healthcare and mental health applications, false positives are often tolerated because they err on the side of caution—better to investigate a harmless case than to overlook a serious one.

False Negatives

False negatives happen when stressful texts are misclassified as non-stressful. This is far more severe in the mental health context because it means failing to detect someone who may be in distress or even expressing suicidal ideation. Missing these signals can prevent timely intervention and support, which could have life-threatening consequences. Therefore, minimizing false negatives is critical, even if it means accepting more false positives.

Confusion Matrix Context

Looking at the confusion matrix, **false positives (97) are significantly higher than false negatives (47)**. Since false negatives are the more severe error type in this domain, it is fortunate that they are not the dominant error here. However, the number of false negatives is still significant and in a mental health setting, even a single missed case can be concerning. This highlights that further tuning should focus on reducing false negatives to ensure that stressful texts are reliably detected.

Exploring More About The Misclassified Data

Now given that there are still quite a number of false positive and false negative cases where the Hybrid RoBERTa model has misclassified, the next step is to explore these errors in greater depth.

1. This means going beyond the raw counts and examining which subcategories of texts are most prone to misclassification. For example, stressful texts could include categories like academic pressure, workplace stress or personal relationships and each may have different misclassification rates. Understanding the percentage of errors within each subcategory will help identify whether the model struggles more with certain themes of stress.

2. Equally important is analyzing the raw text instances where misclassification occurs. By reviewing the actual misclassified samples, we can uncover patterns such as ambiguous wording, sarcasm or subtle emotional cues that the model fails to

capture. This qualitative inspection provides context that pure numbers cannot and it can guide feature engineering or the use of more advanced language models that better understand nuance.

3. Examining the relationship between text length and misclassification rates can reveal structural weaknesses in the model. Short texts may lack sufficient context, making them harder to classify correctly, while very long texts may introduce noise that confuses the classifier. By quantifying how misclassification varies with text length, we can determine whether preprocessing strategies (like segmenting long texts or enriching short ones with contextual embeddings) could improve performance.

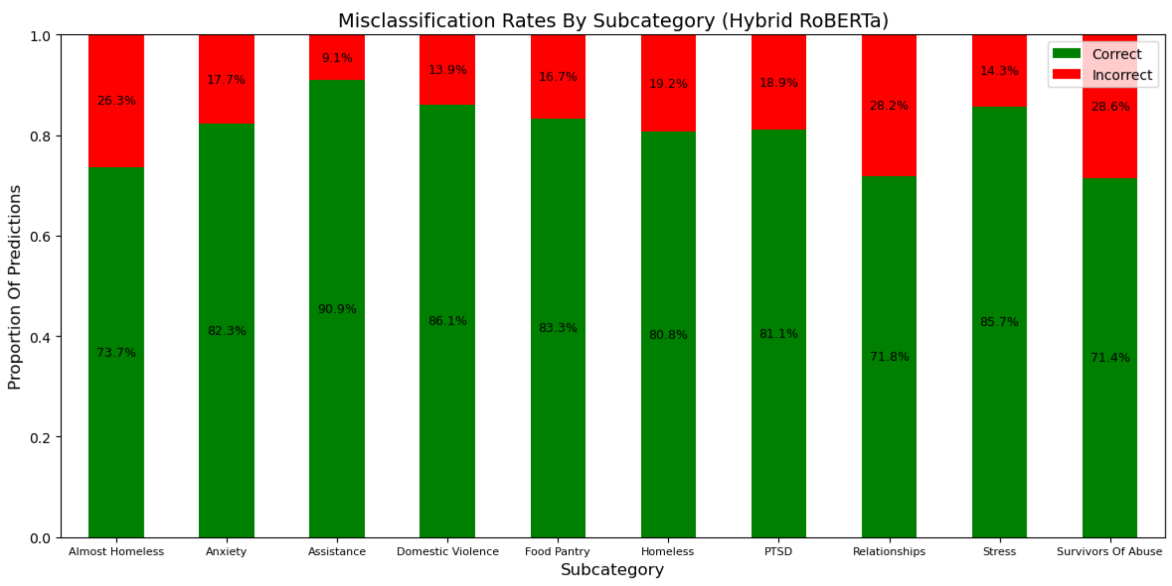
4. Finally, we will explore the words in the misclassified texts more in-depth using model interpretability methods including SHAP and LIME.

Exploration #1: Misclasification Rates Of Test Data By Subcategory

Using the `dreaddit_test` dataset, we will analyze the misclassification rates of the Hybrid RoBERTa Model, grouped by subcategory. The analysis will be visualized as a **stacked bar chart**, where each bar represents a subcategory normalized to 100%. Correct predictions will be shown in green, while misclassifications will be shown in red, allowing us to easily compare performance across different subcategories.

In [284…

```
# Missclassification By Subcategory
test_analysis = dreaddit_test.copy()

test_analysis['Predicted'] = predictions
test_analysis['Correct'] = test_analysis['Label'] == test_analysis['Predicted']

# Aggregate by subcategory
subcat_summary = test_analysis.groupby('Subcategory')['Correct'].value_counts(nc
subcat_summary = subcat_summary.reindex(columns=[True, False], fill_value=0)

# Plot stacked bar chart
ax = subcat_summary.plot(kind='bar', stacked=True, figsize=(12, 6), color=['gree
for idx, row in enumerate(subcat_summary.itertuples(index=False)):
    correct_pct = row[0] * 100
    incorrect_pct = row[1] * 100
    if row[0] > 0:
        ax.text(idx, row[0]/2, f"{correct_pct:.1f}%", ha='center', va='center',
    if row[1] > 0:
        ax.text(idx, row[0] + row[1]/2, f"{incorrect_pct:.1f}%", ha='center', va

# Customize x-axis labels
ax.set_xticklabels(["Almost Homeless", "Anxiety", "Assistance", "Domestic Violer
                    "Stress", "Survivors Of Abuse"], fontsize=8)

plt.title('Misclassification Rates By Subcategory (Hybrid RoBERTa)', fontsize=14
plt.ylabel('Proportion Of Predictions', fontsize=12)
plt.xlabel('Subcategory', fontsize=12)
plt.legend(['Correct', 'Incorrect'], loc='upper right')
plt.xticks(rotation=0)
plt.ylim(0, 1)
plt.tight_layout()
plt.show()
```



A key observation is that some categories such as **Assistance**, **Domestic Violence** and **Stress** achieve high classification accuracy rates of over 85% while others like **Almost Homeless**, **Survivors of Abuse** and **Relationships** show higher misclassification rates at around 25% to 30%. Most categories fall within the 15 to 20% error range, suggesting that while the model performs reasonably well overall, certain types of stressful contexts are more challenging to classify correctly.

One possible reason for higher misclassification in some categories is the **complexity and overlap of language used in these contexts**. Texts in these categories often contain nuanced emotional cues, indirect references to stress or mixed signals that make them harder for a model to distinguish. **Relationships and Domestic Violence** involve context-dependent language, which can be difficult for a classifier to interpret without deeper semantic understanding.

In the mental health context, these misclassifications highlight the importance of refining the model to better capture subtle linguistic features. Categories with lower error rates may be easier to classify because the language used is more concrete and specific, reducing ambiguity. On the other hand, categories tied to **trauma, abuse or housing insecurity** often involve complex narratives or overlapping themes which increase the likelihood of misclassification.

Exploration #2: Analyzing Raw Text Of Misclassification Instances

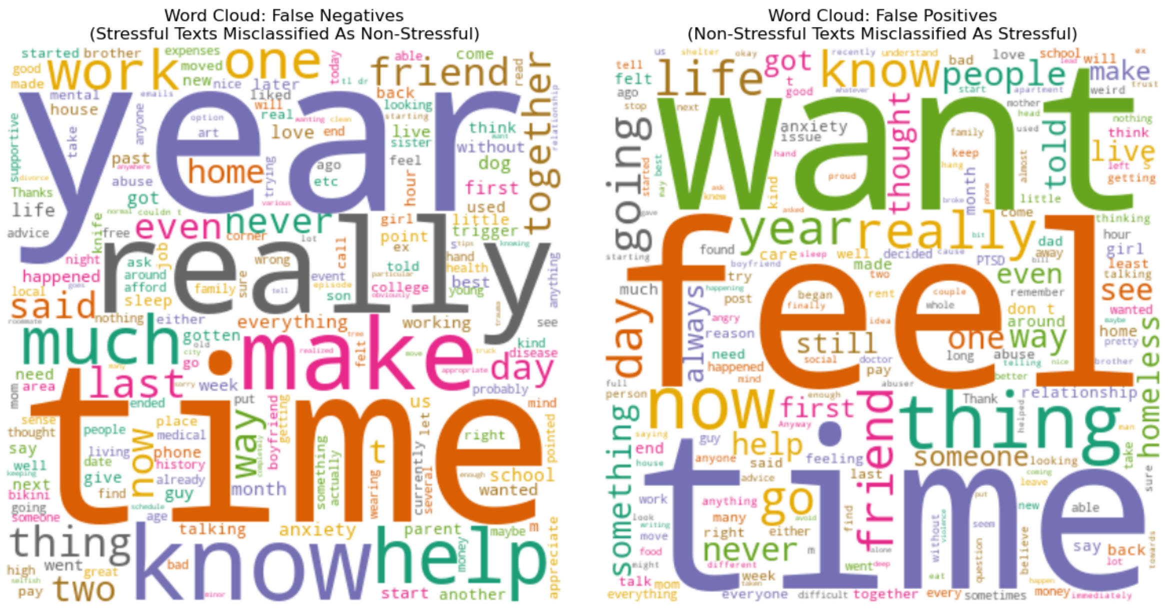
In this exploration, we aim to analyze the raw text instances that were misclassified by the Hybrid RoBERTa Model. Misclassifications are divided into two categories:

- **False Negatives (FN):** Texts that are actually stressful but were classified as non-stressful by the model
- **False Positives (FP):** Texts that are non-stressful but were classified as stressful

Among these, false negatives are of higher concern as failing to identify stressful texts could lead to unaddressed mental health risks. By examining the content of these misclassified texts, we can identify recurring words, phrases and patterns that the model struggles to capture. This analysis will be performed using **Word Clouds for visual**

inspection and **n-gram frequency analysis** to understand common sequences of words. The insights gained may inform future improvements in feature engineering, preprocessing and model design.

```
In [286…
# Word Cloud Analysis For Misclasified Texts
test_analysis = dreaddit_test.copy()

test_analysis['Predicted'] = predictions
false_negatives = test_analysis[(test_analysis['Label'] == 1) & (test_analysis['
false_positives = test_analysis[(test_analysis['Label'] == 0) & (test_analysis['

# Generate WordCloud objects
wc_fn = WordCloud(width=400, height=400, background_color='white', colormap='Dar
wc_fp = WordCloud(width=400, height=400, background_color='white', colormap='Dar
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# False Negatives
ax1.imshow(wc_fn, interpolation='bilinear')
ax1.axis('off')
ax1.set_title("Word Cloud: False Negatives\n(Stressful Texts Misclassified As No

# False Positives
ax2.imshow(wc_fp, interpolation='bilinear')
ax2.axis('off')
ax2.set_title("Word Cloud: False Positives\n(Non-Stressful Texts Misclassified A

plt.tight_layout()
plt.show()
```



Similarities In The Word Clouds:

Both false negatives and false positives prominently feature words like year, time, day, want, know, friend, feel, home and anxiety. This suggests that certain high-frequency, contextually ambiguous words are consistently problematic for the classifier. Additionally, words like feel and anxiety appear in both, showing that emotional language is difficult to disambiguate between stressful and non-stressful contexts.

Differences In The Word Clouds:

False Negatives (Stressful Text Misclassified As Non-Stressful):

- Words like help, need, job, life, home and anxiety dominate
- These often signal urgency, distress or requests for support
- The model likely underestimates stress when these words appear in casual or neutral contexts

False Positives (Non-Stressful Misclassified As Stressful):

- Words like never, people, think and now are more prominent
- These are reflective or general life expressions that may not indicate stress
- The model likely overestimates stress when encountering abstract or philosophical language

Possible Insights:

The Hybrid RoBERTa model tends to overweight common words (time, day, year) that appear in both stressful and non-stressful texts, reducing its discriminative power. Meanwhile, words like help, need, job and life are strong indicators of stress but their signal is diluted because they also occur frequently in non-stressful contexts. Abstract words such as never, people and think may be misinterpreted as stress markers due to their co-occurrence with stressful contexts in the training data.

Now let us look at the most common phrases that the Hybrid RoBERTa model has misclassified. We will use a combination of Bigrams and Trigrams to look at the top 20 most common phrases for both categories.

In [288…

```python
# N-gram Analysis for Hybrid RoBERTa
from IPython.display import display, HTML
from sklearn.feature_extraction.text import CountVectorizer

test_analysis = dreaddit_test.copy()
test_analysis['Predicted'] = predictions
false_negatives = test_analysis[(test_analysis['Label'] == 1) & (test_analysis['
false_positives = test_analysis[(test_analysis['Label'] == 0) & (test_analysis['

# Function to extract top n-grams
def get_top_ngrams(text_series, ngram_range=(2,3), top_n=20):
    vectorizer = CountVectorizer(ngram_range=ngram_range, stop_words='english')
    X = vectorizer.fit_transform(text_series)
    sum_words = X.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vectorizer.vocabula
    words_freq = sorted(words_freq, key=lambda x: x[1], reverse=True)
    return words_freq[:top_n]

# Get top n-grams for False Negatives and False Positives
top_ngrams_fn = get_top_ngrams(false_negatives['Text'])
top_ngrams_fp = get_top_ngrams(false_positives['Text'])
df_fn_ngrams = pd.DataFrame(top_ngrams_fn, columns=['N-gram', 'Frequency'])
df_fp_ngrams = pd.DataFrame(top_ngrams_fp, columns=['N-gram', 'Frequency'])

# Function to display two DataFrames side by side
def df_to_html_side_by_side(df1, df2, title1="", title2=""):
    html = f"""
    <div style="display: flex; justify-content: space-around;">
        <div style="margin-right: 20px; text-align: center;">
            <h4>{title1}</h4>
            <div style="display: inline-block; text-align: left;">
                {df1.to_html(index=False)}
            </div>
        </div>
        <div style="text-align: center;">
            <h4>{title2}</h4>
            <div style="display: inline-block; text-align: left;">
                {df2.to_html(index=False)}
            </div>
        </div>
    </div>
    """
    display(HTML(html))

# Display the tables side by side
df_to_html_side_by_side(df_fn_ngrams, df_fp_ngrams, title1="False Negatives (Str
                        title2="False Positives (Non-Stressful Texts Misclassifi
```

False Negatives (Stressful Texts Misclassified As Non-Stressful)

N-gram	Frequency
don know	3
tl dr	3
really appreciate	3
ve gotten	3
past weeks	3
ll probably	2
make sure	2
don mind	2
time just	2
financial industry	2
pay bills	2
years ago	2
little year	2
got pregnant	2
time got	2
said couldn	2
best friend	2
developed anxiety	2
high school	2
knife hand	2

False Positives (Non-Stressful Texts Misclassified As Stressful)

N-gram	Frequency
feel like	9
don know	6
don want	6
just don	4
felt like	3
feel proud	3
best friend	3
feeling like	3
just wanted	3
mental health	3
don work	3
thing really	3
time life	3
need help	3
years ago	3
long time	3
pay rent	3
junk food	2
fast food	2
know feel	2

Overlap Between False Negatives and False Positives:

- Shared n-grams: don know, best friend appear in both categories.
- These phrases are highly context-dependent and can signal either stress or casual conversation, making them difficult for the model to classify correctly

False Negatives (Stressful Texts Misclassified As Non-Stressful):

Many phrases often reflect emotional or situational distress but were missed by the model. Expressions like tl dr and 11 probably may have led the model to underestimate stress because they resemble neutral or informal language. Phrases like past weeks and years ago suggest ongoing issues but may not have been weighted strongly enough.

False Positives (Non-Stressful Texts Misclassified As Stressful):

Phrases like mental health and need help are strong stress-related terms but in some contexts they may appear in informational or casual discussions rather than personal distress. Most of the phrases are not inherently stressful but were misclassified. Some phrases may have been over-weighted as stress markers due to co-occurrence with stressful contexts in training data.

Possible Insights:

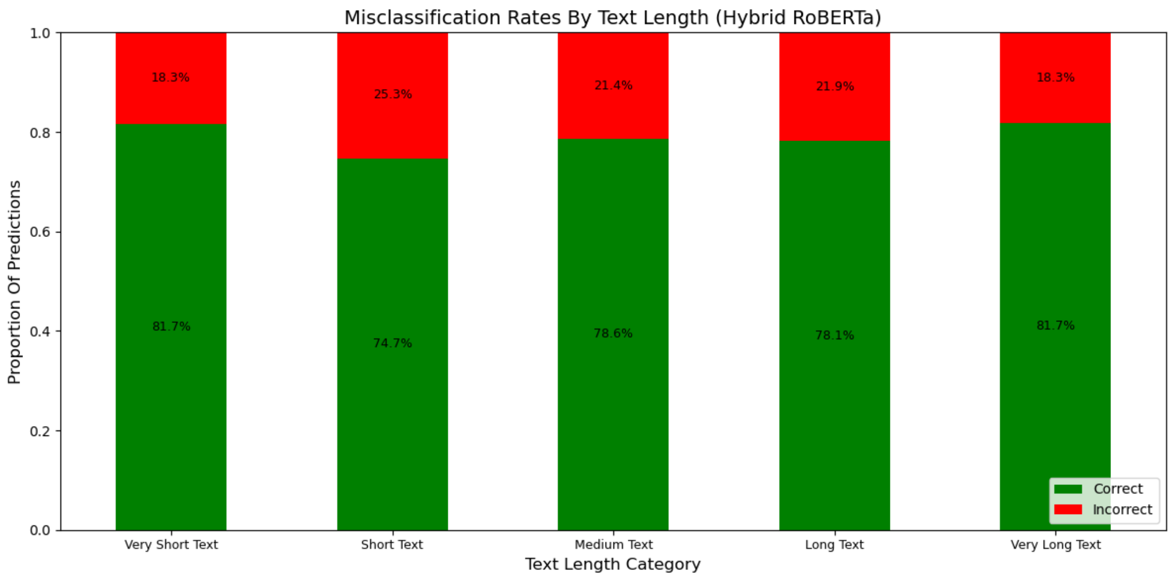
The same n-gram can appear in both stressful and non-stressful texts, showing the model struggles with semantic nuance. Terms like anxiety disorder or social anxiety trigger false positives, suggesting the model equates clinical terminology with stress regardless of context. Phrases like want talk, single mom and mental energy are strong indicators of stress but may be overlooked when surrounded by neutral or casual language. Words like weeks ago, years ago and long time are weak signals of stress but appear frequently in misclassifications, showing the model may overweight them.

Exploration 3: Misclassification Rates By Text Length Category

In this exploration, we will examine the impact of text length on misclassification rates produced by the Hybrid RoBERTa Model. During the data preprocessing stage, texts were categorized into five groups based on length: **Very Short Text, Short Text, Medium Text, Long Text, and Very Long Text**. The analysis will evaluate misclassification rates across these categories, with results presented using a stacked bar chart for clarity.

In [290…

```python
# Misclassification Analysis by Text Length (Hybrid RoBERTa)

# Copy test data
test_analysis = dreaddit_test.copy()

# Use predictions from Hybrid RoBERTa evaluation loop
test_analysis['Predicted'] = predictions   # predictions from your RoBERTa model

# Determine correctness
test_analysis['Correct'] = test_analysis['Label'] == test_analysis['Predicted']

# Aggregate by text length category
length_summary = test_analysis.groupby('Text Length')['Correct'].value_counts(no

# Ensure both True and False columns exist
length_summary = length_summary.reindex(columns=[True, False], fill_value=0)

# Plot stacked bar chart
ax = length_summary.plot(kind='bar', stacked=True, figsize=(12, 6), color=['gree

# Add percentages inside the bars (centered in each bar segment), only if >0
for idx, row in enumerate(length_summary.itertuples(index=False)):
    correct_pct = row[0] * 100
    incorrect_pct = row[1] * 100
    # Correct (green)
    if row[0] > 0:
        ax.text(idx, row[0]/2, f"{correct_pct:.1f}%", ha='center', va='center',
    # Incorrect (red)
    if row[1] > 0:
        ax.text(idx, row[0] + row[1]/2, f"{incorrect_pct:.1f}%", ha='center', va
# Customize x-axis labels
ax.set_xticklabels(["Very Short Text", "Short Text", "Medium Text", "Long Text",

plt.title('Misclassification Rates By Text Length (Hybrid RoBERTa)', fontsize=14
plt.ylabel('Proportion Of Predictions', fontsize=12)
plt.xlabel('Text Length Category', fontsize=12)
plt.legend(['Correct', 'Incorrect'], loc='lower right')
plt.xticks(rotation=0)
plt.ylim(0, 1)
plt.tight_layout()
plt.show()
```



The chart provides a clear view of how text length influences the model's ability to classify stressful versus non-stressful texts. Several important patterns emerge. For the overall trends:

- Very Long Texts show a high accuracy (18.3% incorrect), suggesting that longer inputs provide richer linguistic cues for the model to leverage
- Short Texts stand out with the highest misclassification rate (25.3% incorrect), indicating that this category poses the greatest challenge.
- Very Short Texts surprisingly perform significantly better than Short Texts, with only 18.3% misclassified.
- Medium and Long Texts fall in between, with moderate misclassification rates of about 22%.

Why Short Texts Are Most Problematic:

Short texts often contain enough words to introduce ambiguity but not enough context for the model to resolve it. For example, a short sentence like "I feel anxious" may be flagged as stressful, but "I feel anxious about exams" provides clearer context. The intermediate length of short texts means they can include emotionally charged words without sufficient surrounding detail, leading to confusion. In essence, they are long enough to contain misleading signals but too short to provide clarifying context.

Why Very Short Texts Perform Better:

Very short texts (eg. single words or brief phrases) often carry direct, unambiguous signals. In mental health datasets, these may include clear markers like "help", "anxiety" or "depressed". Because they are concise, the model can more easily map them to stress-related categories without being distracted by additional neutral or ambiguous language. This clarity reduces misclassification rates compared to short texts, which may mix stress-related terms with everyday expressions. In sensitive domains like medical and mental health, very short texts often correlates with urgency or explicit distress, making these signals easier for the model to detect correctly.

Key Insight:

The model struggles most when texts are short but not minimal as these contain enough variability to blur the distinction between stressful and non-stressful contexts. Very short texts in contrast tend to be sharper indicators of stress, while longer texts provide the model with richer context to make more accurate predictions. This highlights the importance of context length in classification tasks, especially in domains where subtle differences in phrasing can carry significant meaning.

Exploration 4: Model Interpretations

For model interpretations, we will be **making use of LIME and SHAP** for the Hybrid RoBERTa model.

LIME For Local Explanations:

LIME (Local Interpretable Model-Agnostic Explanations) is a post-hoc interpretability technique that explains individual model predictions by approximating the model locally with a simpler, interpretable model. For text classification tasks, LIME perturbs the input text by removing or altering words and observes how the model’s predictions change. By assigning importance weights to each word, LIME identifies which tokens contribute most positively or negatively to the predicted class. This allows practitioners to understand why the model made a particular prediction, uncover biases, and detect potential failure modes — all without modifying the model itself.

LIME is particularly suitable for transformer-based models like RoBERTa because these models are highly complex and inherently opaque. RoBERTa leverages deep attention mechanisms and contextual embeddings, making it difficult to directly interpret the effect of individual words. LIME provides a way to "open the black box" locally for each prediction, by approximating RoBERTa’s decision boundary in the neighborhood of a specific input. This allows one to understand which words influenced the model’s classification without needing to analyze the full high-dimensional embedding space or attention layers directly.

Step 1: Selecting Some Misclassified Examples

The first step in using LIME conceptually is to identify misclassified instances from the model’s predictions. For example, a particular text has a true label of Stress but RoBERTa predicted Non-Stress. Focusing on misclassified examples helps reveal patterns in model errors, such as over-reliance on certain words or failure to capture context.

Similarly, we will break the misclassifications into two classes - **False Positives (Predicted = 1, Reality = 0) and False Negatives (Predicted = 0, Reality = 1)**.

Step 2: Examine LIME Word Contributions

LIME analyzes the text by perturbing it and measuring how the prediction changes. Conceptually, it assigns weights to each word indicating whether it pushes the prediction toward the predicted class or the true class. For the example above, the contributions might be:

- Words pushing toward Non-Stress: "manage", "think", "I'll"
- Words pushing toward Stress: "haven't", "slept", "days"

This breakdown allows a structured, token-level understanding of the misclassification. For this step, we will analyze the words and phrases of the top few rows of text that the Hybrid RoBERTa model has missclassified with a high level of confidence.

Texts Classified Under False Positives:

We will identify instances where the model predicted a positive label incorrectly, focusing only on cases where the model was highly confident in its mistaken prediction (confidence > 0.8). This is used to identify the commonly used text words that might lead to the model predicting that the text is stressful although in reality, the text is not a stressful one.

In [292…

```
# Filter misclassified rows
misclassified = test_analysis[test_analysis['Correct'] == False][['Text', 'Predi
    'Predicted': 'Predicted Label', 'Label': 'True Label'})

# Join with the original dreaddit_test to get Confidence
misclassified_joined = misclassified.merge(dreaddit_test[['Text', 'Confidence Le

# Filter for False Positives: Predicted = 1, True = 0 AND Confidence > 0.8
false_positives_high_conf = misclassified_joined[(misclassified_joined['Predicte
                                                 (misclassified_joined['True Lab
                                                 (misclassified_joined['Confider

# Sort by Confidence in descending order
false_positives_high_conf = false_positives_high_conf.sort_values(by='Confidence

# Increase display width for Text column
pd.set_option('display.max_colwidth', 1000)
false_positives_high_conf
```

Out[292…

	Text	Predicted Label	True Label	Confidence Level
1	So, I've been homeless since about the first, but I was expecting this. What I wasn't expecting was the changes I've made in my life since becoming homeless. Thankfully I have a job, so I do have money. But I hate everywhere that's affordable to eat! I've been a big junk food/fast food fan for years, so I'm a bit overweight, but lately, I just don't seem to care for meat as much, and fast food is just so.. gross right now.	1	0	1.000000
13	I just wanted to say thank you to everyone that posts on here. I just found this subreddit yesterday and reading all of your guys' posts have made me feel so much less alone. Knowing that other people understand what I'm going through for some reason has helped me tremendously. Particularly a post that someone made about feeling like their trauma isnt real and like convincing yourself that it didnt happen. I have felt like I'm crazy for years because I was in such deep denial about what happened to me.	1	0	1.000000
21	I'd never heard this idea until recently, and I'm starting to see it cropping up more, always in opposition to any kind of support (or even empathy) for the homeless - any attempt to help the homeless is pointless because 'most' homeless people want to be homeless; therefore housing programs/assistance, mental health & substance abuse support are wasted because the homeless are too lazy to get themselves out of homelessness and 'enjoy' their condition, because they don't have to work, because they don't have to pay bills, because they don't have responsibilities, because they're lazy etc. I strenuously disagree with this, and I feel like it's an attempt to move the goalposts; there's a growing awareness that homeless people may not necessarily be homeless because they got themselves into that position and they 'deserve' it, but because of substance abuse issues and mental health problems, coupled with a breakdown in social network. But that's a lie, or 'lefty propaganda', for some ...	1	0	1.000000
22	I have a question about my ex who has a past of violence against women. I was never warned about it but I found out he was violent and I left. His ex has a full life restraining order against him.Now he is on probation for assaulting a police officer for 3 years in the past year he has gone to jail three times for domestic violence. His latest trip to jail was last week for domestic violence his third time. I was wondering what do you think his punishment will be since he's not learning his lesson from the punishments given to him and he just doesn't care.	1	0	1.000000

	Text	Predicted Label	True Label	Confidence Level
48	I feel like I am just being written off because of my diagnosis before anyone even tries to look into it and see if anything else might be at play. I'd just as soon not say anything about my PTSD to be taken seriously, but it appears in my medical records. Or when I list my medications as required, Drs usually ask me what the meds are treating and it comes up then. Has anyone else had this experience? What has worked for you to be taken seriously?	1	0	1.000000
61	I've been renting since university and I'm getting sick of seeing the same amount of money I'd pay towards a mortgage go into the pocket of someone else. I have a junior role in the company, it's going somewhere (I hope) but not fast. I find financially I'm staying basically cash-neutral and my quality of life isn't brilliant. I'm not managing to save anything. I've been thinking that I should get a house with the money I inherited.	1	0	1.000000
73	If he's the textbook abuser, she is the textbook victim. She keeps giving him chances and accepting his apologies and living in this cycle of abuse. She thinks she's the one doing something wrong. I keep telling her that the only thing she is doing wrong is staying with this guy and thinking he will change. I tell her she does not deserve this treatment.	1	0	1.000000
85	Perhaps it is weird to process something after such a long time, but not really. I always felt strong at how I managed to leave, but I never had any closure. I got out of the relationship and pretended nothing ever happened. Just wanted to share I had an eye-opening experience tonight and I feel at peace about everything that happened now because I have deep down, stopped blaming myself. Looking back at every experience I had with him, the mental torture I went through, I cannot believe what I dealt with, but now finally feel ok.	1	0	1.000000
91	I was raised by a narcissistic grandmother and emotionally unavailable mother. In fact, when I was born, she tried to take it upon herself to take full custody away from my mother because apparently my mom was 'unfit' to be a mother. She never did because she found out that my biological father might have had to be involved. My grandma kept me away from my father my entire life, he barely knows I exist, I've never spoken directly with him, in fact, his identity was hidden from me for 22 years, up until last May. My grandmother was very emotionally abusive towards me as a child.	1	0	1.000000

	Text	Predicted Label	True Label	Confidence Level
95	*Begins quoting a rap and making hand gestures* "Go out and make a name or do something!" Everyone ignores him. He came back to his bunk and starts mumbling, "I hear you talking under your breathe man don't think I don't cause I do don't start with me today." Not sure if that was directed towards me or not because I never did that, but whatever. He starts to leave and I start going,	1	0	1.000000
100	I have been seeing mia for about 3-4 months Last night we were together drinking at a little get together before going to the club about 7-8 people in a small group, everyone knows each other pretty well. Mia was drunk and being quite obnoxious which can be fun just as much as it can be bad? She started talking about how her girl friends all get nice things from their boyfriends, a $500 hair job, presents/gifts whatever Made me feel a bit uncomfortable because it was clearly something targeted at me and she was saying this in front of everyone	1	0	1.000000
117	My flaws seemed huge to me, and I assumed everyone thought negatively of me all the time. My life was consumed by a spiral of negative thoughts and social anxiety. I went to my doc, admitted my social anxiety, and he prescribed me Lexapro 10mg/day. I took my first pill a few days ago, and not two hours later, I had nearly complete relief of my anxiety of all kinds. The bad thoughts completely stopped.	1	0	1.000000
81	Now that I’m single I find that I don’t have much friends I can hang out with. With the few guy friends I have we all just play sports together and that’s literally it, nothing outside of that. No one I can really be like “hey let’s go to this concert” or “Yo, let’s hit this party up”. I also feel like I’m kind of too mature for ppl my age, which is conflict sometimes too. I also genuinely want some gal friends and nothing romantic but just some girls to hang out with but I feel all girls just assume all guys have bad intentions or just want to get in their pants.	1	0	0.833333

	Text	Predicted Label	True Label	Confidence Level
127	I then confronted my parents and finally the walls came tumbling down. It wasn't the first time I asked them about it, I did so on at least 3 different occasions throughout my twenties. Well after 2 weeks I got written confirmation, when my parents handed me a note from 1986, where they wrote down, what I told them, when I talked about it afterwards again and again. I addition to this, the memories came back from when I was about 12-15, when my parents used to lock me up in my room and bathroom and made me learn my latin vocabularies on my knees, only being allowed to eat, when I was finnished. That was something I kinda always knew happened, but I thought it was just on a few occasions and well - I was such a rascal, how else could they have handled me?	1	0	0.833333

Key Observations from the False Positive Texts:

Words and phrases frequently appearing include:

- **Positive/Neutral Framing:** "thank you", "feeling less alone", "peace", "manageable", "helped me tremendously"
- **Action Words Or Self-Agency:** "managed", "confronted", "stopped blaming myself", "getting sick of"
- **Words Describing Experiences Rather Than Explicit Stress:** "party", "club", "friends", "concert"

Explicit indicators of stress, trauma or sensitive experiences like "homeless", "PTSD", "violence", "abuse" and "anxiety" are present but often embedded in long sentences with mitigating or neutralizing context.

Word Importance Analysis With LIME:

Using LIME, each high-confidence false positive can be analyzed by perturbing individual words to see their effect on the model's prediction. In these posts, words such as "homeless", "PTSD", "abuse" and "violence" would likely receive strong weights pushing the prediction toward Stress, while words expressing coping, reflection or gratitude such as "managed", "thank you" and "peace" would carry smaller influence. LIME highlights which words the model relied on most, revealing that early stress-related tokens are over-weighted, while positive or resolving phrases later in the text fail to sufficiently counterbalance the model's prediction.

Why The Hybrid RoBERTa Model Misclassified:

The Hybrid RoBERTa model misclassified these posts because it overemphasized stress-related words, even when the posts' overall sentiment was neutral or positive. Many posts describe difficult experiences but conclude with coping, gratitude or self-reflection which the model underestimates. Because transformer attention often prioritizes strong stress-indicating tokens at the start of the text, the model confidently predicts Stress, resulting in high-confidence false positives despite the true label being Non-Stress.

Insights and Implications For Error Analysis:

These misclassifications reveal a bias in the model toward over-predicting stress when sensitive or challenging topics are present. LIME would show that words associated with stress carry more weight than words signaling resolution or coping, explaining why the model is confidently wrong. This suggests potential improvements, such as giving more attention to context and concluding phrases, better handling of long posts or emphasizing sentence-level embeddings that capture both distress and resolution.

Now we will perform the exact same analysis and insights on the False Negative texts.

Texts Classified Under False Negatives:

We will identify instances where the model predicted a negative label incorrectly, focusing only on cases where the model was highly confident in its mistaken prediction (confidence > 0.8). This is used to identify the commonly used text words that might lead to the model predicting that the text is non-stressful although in reality, the text is a stressful one.

NOTE: For the False Negative scenarios, the condition of the Confidence Level has changed from > 0.8 to >= 0.8 to include text where the confidence level is exactly at 0.8. If set to > 0.8, there will only be 2 rows of output which might not be sufficient for analysis.

In [294…
```
# Filter misclassified rows
misclassified = test_analysis[test_analysis['Correct'] == False][['Text', 'Predi
    'Predicted': 'Predicted Label', 'Label': 'True Label'})

# Join with the original dreaddit_test to get Confidence
misclassified_joined = misclassified.merge(dreaddit_test[['Text', 'Confidence Le

# Filter for False Negatives: Predicted = 0, True = 1 AND Confidence >= 0.8
false_negatives_high_conf = misclassified_joined[(misclassified_joined['Predicte
                                                 (misclassified_joined['True Lab
                                                 (misclassified_joined['Confider

# Sort by Confidence in descending order
false_negatives_high_conf = false_negatives_high_conf.sort_values(by='Confidence

# Increase display width for Text column
pd.set_option('display.max_colwidth', 1000)
false_negatives_high_conf
```

Out[294…

	Text	Predicted Label	True Label	Confidence Level
64	Then I came home. My Mom pointed it out first, I went from being the class clown and the life of the party, to being the quiet guy who stood in the corner of the room. I went from a musician and avid gamer, to having no interest in any of it, and no replacement hobby. The things I had the most passion for in life were gone. It was like someone removed one of my five senses and my soul at the same time.	0	1	1.0
119	Alright so, for my entire life I've been a straight-A student. High honor roll every time. I never expected any less and neither did my parents, teachers, friends, etc. Over the past year (8th grade) I've developed anxiety, depression, and OCD. (Not self diagnosed.)	0	1	1.0
37	We've funnelled about 4k into saving our dog's life. As you can see- we haven't gotten that in donations, but what we've gotten has really saved our asses. I found out my dog had a 'treatable' disease right after graduating from college. The college debt was already a burden, and after my partner got laid off- the panic set in that we would have to put my best friend down. He's too young for this, and has been too good to us.	0	1	0.8
47	She's the first person I've ever really opened up to. I haven't told her everything about whats happened, but she does know about my anxiety (which I get from my PTSD) and she reacts sportively to it. To some extent, I let me be "myself" around her, whatever I am. She's moving. She's moving to Maryland.	0	1	0.8
80	And yes, I would love help. I live in <location>. I can't use a phone as we only have one; house phone, that mom is always sitting near. I would love to be given links to sites that can help my brother and I to places that will take us. Because today was the last straw with my 'mother' hitting my brother with a tennis racket as my 'dad' restrained him down on the couch.	0	1	0.8
84	She also wasn’t wearing her ring. She was also extremely upset when I grabbed her phone, which is locked with a password I don’t know, to hand it to her. Well on Monday we barely spoke during the day, and she was very quiet at dinner. Dreading what might come but knowing I had to ask I asked her if everything was ok. She said no, that she couldn’t marry me, couldn’t have kids as she would be a bad mother, and couldn’t see us being together anymore.	0	1	0.8

	Text	Predicted Label	True Label	Confidence Level
90	And here I am, several years later. But I hadn't expected to still be alive by now. So planning anything for more than a few hours in the future feels stupid and pointless. Self-harm and suicide seem like pretty decent options most days. As a compromise, I sleep, because being unconscious gives me a chance to not be in this world for a little while.	0	1	0.8
97	First off I'm male but my relationship with the woman is strictly platonic stemming from professional, we are co-workers. It's no secret to anyone that she's in an abusive relationship as she's come to work several times over the last several months with blackeyes etc and admits to being abused. She been in the relationship for 1.5 years which turned info is out of the way, here's the new twist that has brought me to split roads. Yesterday she showed me a text from him detailing how angry he'd been at her 12 yr old and that if he didn't have so much will power he would have choked her son to death and that she needs to get her son to behave so he doesn't have to crack his skull. I encouraged her to show that to Leo and have him removed from the apartment as well as get an order of protection, she seemed to be considering that but has expressed fear of retaliation in the past.	0	1	0.8
103	But what should I say? Part of me wants to tell her I'm sorry for being a shit boyfriend at the start and that as I've gotten older I've gotten wiser and more experience. But would she even care about that shit? Or should I just start off with "Hey, haven't seen you in awhile, hows everything" and start from there? Ugh, maybe I'm overthinking this... Anyways would really appreciate you guys or gals out there with more experience than me guiding me through this minor plight I put myself in.	0	1	0.8
109	As is usually the case with the stories I read here, I've hit the bottom. I'm at the very last point before pitching a tent in a field, and begging at the street corner for pennies. A lot has happened in the last month, so I'll try to pare it down to make sense and then get to the numbers drama. When I say a lot, I mean we have been homeless since Feb 17th and making it work. There's me (27), my wife (23), and our son (10m).	0	1	0.8

	Text	Predicted Label	True Label	Confidence Level
131	Yep, you read the title correctly. I get anxiety from my dog, more specifically leaving my dog alone for extended periods of time. I got him when he was a puppy and at the time I lived at home with four other humans and my parents two dogs. We all worked different schedules so he wasn't left alone very often, and even when we were all gone there were other dogs their with him. A little over a year and a half ago my bf and I moved in together and my dog had some really bad separation anxiety at first.	0	1	0.8
135	I have GAD with obsessive thoughts (ocd) that are health related (hypochondria) And I am 18, I have been single my whole life, have never had a girlfriend or a kiss or anything of the sort. In elementary school I liked 4 girls throughout my time there and they all just ended up as friends and I never confessed my feelings. In middle school I liked the same girl for 2 years who rejected me. Then I liked two more girls, one who ended up as a friend and another moved. Now in high school I've liked about 11 girls, and no luck at ALL with any of them.	0	1	0.8
138	My mom has a degenerative neurological disease, and can barely walk, bathe, or feed herself anymore. Her most problematic symptom is a movement disorder, but she also has declining mental capacity (but not much confusion/forgetfulness). She either needs to receive in home care or be moved to an assisted living facility as soon as possible. My brother and I made an appointment to have her assessed by the regional health authority at the end of the month. She moved back to her hometown a few years ago, and my brother and I can't take time off work or afford travel, so we can't be there.	0	1	0.8

Key Observations from the False Negative Texts:

Words and phrases that frequently appear include:

- **Explicit Mental Health Indicators:** "depression", "anxiety", "OCD", "PTSD", "self-harm", "suicide", "homeless", "abuse" and "degenerative neurological disease"
- **Severe Life Stressors:** "Financial hardship", "homelessness", "relationship breakdown", "domestic violence", "parental illness", "academic pressure" and suicidal ideation.
- **Long Narrative-Style Posts:** Many texts are lengthy and describe evolving situations rather than short emotional outbursts
- **Mixed emotional tone:** Some posts include problem-solving language ("I encouraged her...", "We made an appointment...", "Would appreciate guidance...") alongside distress signals

Word Importance Analysis With LIME:

Applying LIME-style reasoning to these high-confidence false negatives reveals that words explicitly indicating distress such as suicide, self-harm, anxiety, depression and abuse should push the prediction toward Stress. However, words signaling reflection, planning or advice-seeking such as appointment and encouraged may carry weight toward Non-Stress. LIME would show that the model underweights these strong distress tokens when they are embedded within long narrative posts, while overemphasizing neutral or advisory words. This highlights how the model's attention mechanism distributes influence unevenly across long sequences, causing important stress indicators to have less effect on the prediction.

Why The Hybrid RoBERTa Model Misclassified:

The Hybrid RoBERTa model misclassified these posts because it underestimated the severity of explicit distress signals within long and complex narratives. Despite the presence of clear indicators of mental health issues, domestic abuse or homelessness, the model focused on neutral or advisory portions of the text, interpreting them as signals of emotional regulation rather than stress. Attention mechanisms in the transformer likely prioritized early neutral framing or dispersed focus across the lengthy text, failing to amplify critical stress-related tokens later in the narrative. This results in high-confidence false negatives where genuinely stressed posts are confidently labeled as Non-Stress.

Insights and Implications for Error Analysis:

These false negatives reveal structural weaknesses in the model. Distress signals embedded within long narratives can be diluted while words associated with reflection, planning or advice-seeking may dominate the model's attention, leading to under-prediction of stress. This suggests that sentence-level classification highlighting explicit stress keywords could improve performance. Moreover, the model's reliance on surface tone rather than semantic content indicates a need for domain-adapted fine-tuning and additional attention to clinically relevant language.

SHAP Analysis (For Local Explanations)

SHAP (SHapley Additive exPlanations) is an interpretability framework. It assigns an importance value, called a Shapley value, to each feature to quantify its contribution to a particular prediction. By considering all possible combinations of features, SHAP provides a theoretically sound measure of how much each word pushes the model’s output toward a particular class. Unlike simpler methods, SHAP gives consistent, additive explanations, making it suitable for understanding complex models such as transformers.

Step 1: Identify Misclassified Texts

Start by comparing the model's predictions against the true labels. Separate the misclassified instances into false positives (Predicted = 1, True = 0) and false negatives (Predicted = 0, True = 1).

Step 2: Generate SHAP Values

For each misclassified text, compute SHAP values for all tokens. This involves measuring the contribution of each word to the predicted probability of each class while accounting for interactions with all other words in the sequence. The result is a word-level attribution for each misclassified instance. For the purpose of this project, we will be making use of just 1 False Positive and 1 False Negative instance for analysis.

Step 3: Examine Word Contributions

Analyze which words pushed the prediction toward the predicted class and which words pulled it toward the other class. For false positives, stress-related words may have too much influence while positive or coping words are underweighted. For false negatives, explicit distress words may be underweighted while neutral or explanatory words dominate.

Step 4: Interpret Patterns

Look for systematic patterns across multiple misclassified texts. For example, check whether the model consistently overweights stress words in false positives or underweights them in false negatives. Identify patterns like attention focusing on early tokens, long-text dilution or underestimation of concluding coping language.

In [296…

```
import shap
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from IPython.display import clear_output
clear_output(wait=True)
shap.initjs()

# Step 1: Setup model and tokenizer
model_name = "roberta-base"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)
model.eval()

# Step 2: Select 1 FP and 1 FN text
fp_text = false_positives_high_conf['Text'].iloc[0]
fn_text = false_negatives_high_conf['Text'].iloc[0]

# Step 3: Define logit-difference prediction function
def predict_logit_diff(texts):
    """Return logit difference: Stress - No Stress"""
    if isinstance(texts, (np.ndarray, pd.Series)):
        texts = texts.tolist()
    inputs = tokenizer(texts, padding=True, truncation=True, return_tensors="pt"
    with torch.no_grad():
        outputs = model(**inputs)
        diff = outputs.logits[:, 1] - outputs.logits[:, 0]  # Stress logit minus
    return diff.numpy()

# Step 4: Create SHAP explainer
explainer = shap.Explainer(predict_logit_diff, masker=shap.maskers.Text(tokenize

# Step 5: Compute SHAP values
```

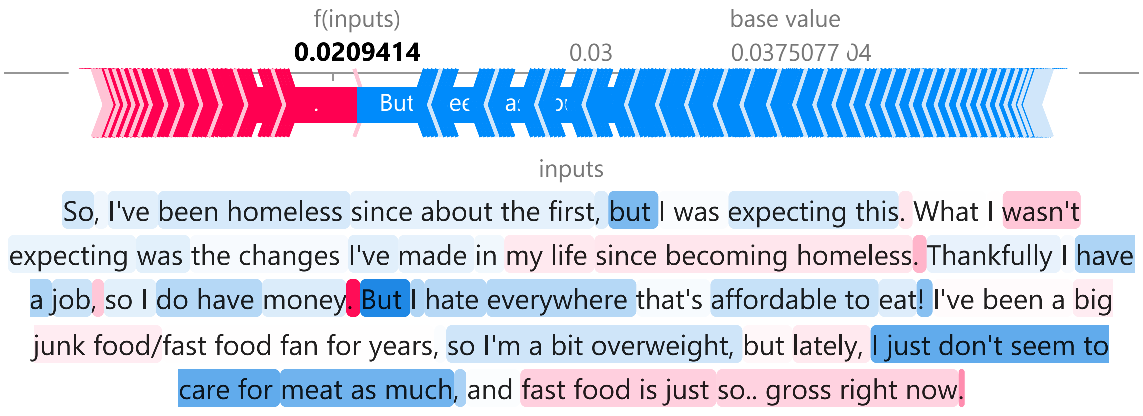
```python
shap_values_fp = explainer([fp_text])
shap_values_fn = explainer([fn_text])
```

```
Loading weights:   0%|          | 0/197 [00:00<?, ?it/s]
  0%|          | 0/498 [00:00<?, ?it/s]
PartitionExplainer explainer: 2it [01:04, 64.01s/it]
  0%|          | 0/498 [00:00<?, ?it/s]
PartitionExplainer explainer: 2it [00:50, 50.27s/it]
```

In [297…

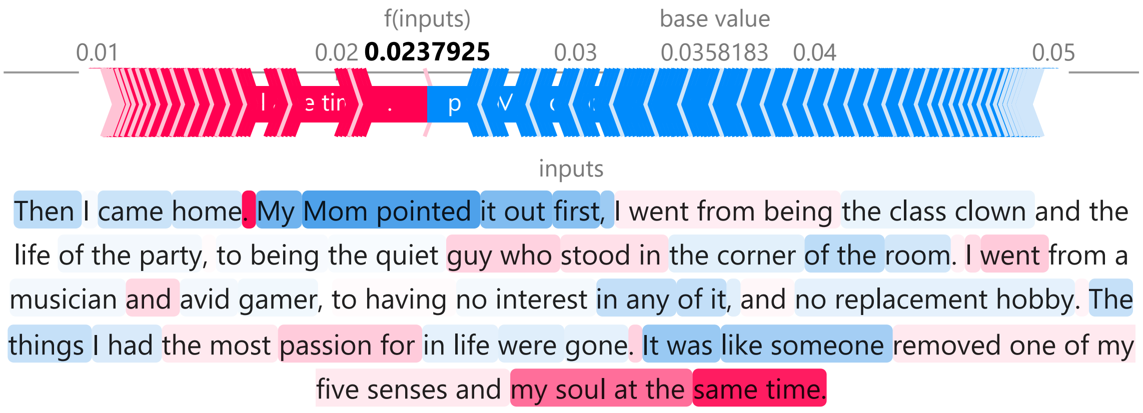
```python
# Step 6: Display SHAP explanations
from IPython.display import display

print("False Positive Example SHAP Explanation:")
print()
shap.plots.text(shap_values_fp[0])
print()
print("\nFalse Negative Example SHAP Explanation:")
print()
shap.plots.text(shap_values_fn[0])
```

```
False Positive Example SHAP Explanation:
```



So, I've been homeless since about the first, but I was expecting this. What I wasn't expecting was the changes I've made in my life since becoming homeless. Thankfully I have a job, so I do have money. But I hate everywhere that's affordable to eat! I've been a big junk food/fast food fan for years, so I'm a bit overweight, but lately, I just don't seem to care for meat as much, and fast food is just so.. gross right now.

```
False Negative Example SHAP Explanation:
```



Then I came home. My Mom pointed it out first, I went from being the class clown and the life of the party, to being the quiet guy who stood in the corner of the room. I went from a musician and avid gamer, to having no interest in any of it, and no replacement hobby. The things I had the most passion for in life were gone. It was like someone removed one of my five senses and my soul at the same time.

How To Read The SHAP Text Plots

Red tokens indicate words that push the model toward predicting Stress (class 1) while Blue tokens indicate words that push the model away from predicting Stress (class 0). The intensity of the color reflects the strength of the contribution. Darker red or blue means the token has a stronger influence on the model output, while lighter shades indicate a weaker influence. The arrows on top of the tokens represent the cumulative effect of each token on the model output. Upward arrows pointing right indicate increasing contribution toward Stress, and the numbers above the arrows show the cumulative probability or logit impact at that position. The base value represents the model's average output before seeing the text. SHAP values show how each token moves the output away from this base value.

False Positive Example (First Text)

The text is about being homeless, diet and personal lifestyle. Red tokens such as "I've been homeless", "the changes" and "fast food" push the model toward predicting Stress. Blue tokens like "I was expecting this", "Thankfully I have a job" and "so I'm a bit overweight" push against Stress. The model predicted Stress (hence a false positive) because it focused heavily on negative or stressful keywords. Positive aspects in the text, such as financial stability and personal reflection are captured as blue, reducing the probability slightly but not enough to avoid the false positive. This example highlights that the model heavily weights negative life events and stress-related words even when the context shows resilience or coping.

False Negative Example (Second Text)

The text discusses identity changes, hobbies, social roles and personal losses. Blue tokens such as "Then I came home" and "my soul at the same time" dominate, pushing the prediction away from Stress. Some red tokens like "of the party" and "musician" provide a slight push toward Stress. The model underestimated stress here because the language is subtle and descriptive, not explicit stress markers. Stress is implied in the narrative through loss of hobbies and changes in social identity, but the model likely focuses on explicit negative keywords. This false negative highlights that the model may miss nuanced stress expressions where negative context is implicit.

Overall, the model tends to assign high importance to explicitly negative or "stressful" words such as homelessness, fast food or personal loss. Potential improvements include incorporating context-sensitive features or fine-tuning the model to detect implicit stress cues such as personal reflection, identity changes and emotional nuance.

Where Would This Model Be Deployed In Real Life?

Hybrid RoBERTa model can be applied in several real-world scenarios where detecting stress in text is valuable. Because this model relies on structured numerical features (such as LIWC and DAL scores) rather than raw text, it is particularly suited for contexts where psychological or emotional markers need to be quantified and analyzed systematically.

1. **Mental Health Hospitals And Counseling Centers:**

   One important application is in clinical settings, where psychologists and counselors record patient conversations or transcribe voice sessions into text. By running these texts through the model, clinicians can quickly identify patterns of stress or emotional distress. This can serve as a supplementary tool to highlight concerning language, helping professionals prioritize cases or monitor changes in patient well-being over time.

2. **Survey-Based Research And Public Health Studies:**

   Another scenario is in large-scale surveys where participants respond to questions about depression, anxiety or stress. Instead of manually coding thousands of responses, the model can automatically classify them into stressful versus non-stressful categories. This allows researchers to efficiently analyze population-level trends, identify risk groups and design targeted interventions.

3. **Online Support Platforms And Forums:**

   In digital communities where individuals share personal struggles (such as Reddit support groups or mental health forums), the model can help moderators flag posts that may indicate high stress or urgent need for support. This ensures timely responses and reduces the risk of critical messages being overlooked. These online platforms are not limited to Reddit but also other social media platforms or websites from the internet where users share their thoughts and feelings.

4. **Workplace Wellness Monitoring:**

   Organizations conducting employee feedback surveys or collecting anonymous comments can apply this model to detect stress-related language. By quantifying stress signals in employee communications, HR departments can gain insights into workplace well-being and address systemic issues before they escalate.

5. **Educational Settings:**

   Schools and universities can use the model to analyze student feedback, essays or reflections for signs of stress. This can help educators identify students who may be struggling emotionally and provide appropriate support services, especially during exam periods or transitions. This is especially important as the age group with the highest rates of stress and depression are in the teenagers and yong adult range (depresion rates are around 7 to 8% in Singapore, teenagers and young adults around 11 to 12%).

Ethical Risks Involved If Model Were To Be Deployed

Deploying a Hybrid RoBERTa model to detect stress in text introduces several ethical risks that must be carefully considered before implementation. These risks are particularly significant in sensitive domains such as healthcare, education and workplace monitoring, where missteps can have serious consequences for individuals privacy, trust and and well-being.

1. **Privacy And Consent Concerns:**

   One of the most pressing issues is the privacy of individuals whose texts are being analyzed. Patients, students or employees may not feel comfortable sharing personal thoughts or stress-related language, especially if they fear judgment or repercussions. Ethical deployment requires explicit consent, transparency about how the data will be used and the assurance that individuals are not coerced into revealing sensitive information. Respecting autonomy is critical and systems should be designed to allow opt-out options without penalty.

2. **Risks of Data Leakage And Misuse:**

   Handling sensitive text data carries the risk of leakage, whether through poor security practices, insider misuse or external breaches. If personal information about stress or mental health were exposed, it could lead to stigmatization, discrimination or even targeted attacks. To mitigate this, strict data governance policies must be enforced, including anonymization, encryption and clear commitments that data will only be used for model training and discarded afterward. Even with safeguards, skepticism may remain so building trust through transparency and accountability is essential.

3. **Subjectivity And Misinterpretation Of Stress Signals:**

   Stress is inherently subjective and text-based indicators can vary widely across individuals. Some people may exaggerate their feelings while others may understate them. This variability can lead to inaccurate predictions, with false positives labeling someone as distressed when they are not and false negatives missing genuine cases of stress. In medical and mental health contexts, such errors could have serious consequences either by overlooking someone in need or by unnecessarily escalating interventions. Ethical deployment requires acknowledging these limitations and ensuring that models are used as supportive tools rather than definitive diagnostic systems.

4. **Bias And Fairness Issues:**

   Language use differs across cultures, age groups and social contexts. A model trained on one demographic may misclassify texts from another, leading to biased outcomes. For example, slang or colloquial expressions might be misinterpreted as stress markers while formal language could mask distress. This raises fairness concerns as certain groups may be disproportionately misclassified. Ethical deployment demands diverse training data, continuous bias audits and careful monitoring to ensure equitable performance across populations.

5. **Over-reliance On Model Eventually:**

   Finally, there is the risk that institutions may over-rely on automated predictions, sidelining human judgment. In contexts like healthcare or counseling, this could reduce the quality of care if professionals defer too much to model outputs. Ethical use requires positioning the model as an aid to human decision-making, not a replacement. Clear guidelines should emphasize that predictions are probabilistic and must be interpreted alongside professional expertise.

Limitations Of Hybrid RoBERTa Model

When considering the limitations of deploying a Hybrid RoBERTa model in healthcare, education or workplace settings, the focus should not only be on technical accuracy but also on the broader societal impact of its predictions. While the model can provide structured insights, there are several reasons why it may not be fully suitable or even potentially detrimental in sensitive contexts.

Impact Of False Negatives: In healthcare, particularly mental health, failing to detect genuine stress signals can have severe consequences. False negatives may mean that individuals experiencing suicidal ideation, PTSD or Major Depressive Disorder (MDD) go unnoticed. In such cases, the absence of timely intervention could lead to worsening conditions or even fatal outcomes. This limitation highlights the ethical weight of deploying a model that cannot guarantee high sensitivity in detecting distress.

Impact Of False Positives: On the other hand, excessive false positives can overwhelm healthcare systems. If the model incorrectly flags non-stressful texts as stressful, clinicians may spend unnecessary time and resources investigating cases that do not require intervention. This misallocation of effort could inflate costs, strain already limited mental health resources and reduce efficiency. In workplace or school settings, false alarms could also stigmatize individuals who are incorrectly identified as stressed, potentially harming trust and morale.

Economic And Resource Implications: From a financial perspective, widespread deployment of such a model could lead to wasted resources if misclassifications are frequent. Healthcare systems already face budget constraints and diverting funds to address false alarms could reduce the availability of care for those truly in need. In schools or workplaces, unnecessary interventions could similarly consume time and money without delivering meaningful benefits.

Despite these limitations, the model does offer efficiency gains. Automated classification can reduce the burden of manual review, which is prone to human error and inconsistency. As a complementary tool, it may help flag potential cases for further evaluation, improving overall monitoring capacity. However, relying too heavily on automation risks sidelining human judgment, which is essential for interpreting complex emotional and psychological signals. The model is best positioned as a supportive tool rather than a standalone solution, complementing human expertise rather than replacing it. This balance ensures that technology enhances care without introducing new risks to individuals or society.

Next Steps - A Possible Future Roadmap (If Solution Were To Be Implemented In Real Life)

Phase 1 – Data Expansion And Collection

The first step is to significantly expand the dataset. Current resources such as `dreaddit_train` and `dreaddit_test` contain only a few thousand rows, which may be insufficient for robust training and evaluation. More diverse and representative data should be collected across different demographics, languages and contexts to ensure the model can generalize effectively. This phase emphasizes careful, ethical data collection with informed consent, ensuring that sensitive mental health information is handled responsibly.

Phase 2 – Model Training And Optimization

Once sufficient data has been gathered, the next phase involves training the Hybrid RoBERTa model. This step is not just about improving accuracy but also about balancing efficiency, runtime and interpretability. Multiple configurations should be tested to identify models with strong performance metrics such as F1-score while maintaining computational feasibility. The goal is to refine the model into a reliable tool that can complement human judgment.

Phase 3 – Controlled Real-Life Experimentation

After optimization, the model should be tested in controlled environments. For example, participants can be divided into two groups: a control group relying solely on human counselors and an active group where counselors are supported by the model's predictions. This setup allows researchers to evaluate how well the model integrates into real-world workflows. In cases where counselor and model predictions differ, confidence levels from both sides should be compared with careful consideration of how much weight to assign to automated versus human judgment.

Phase 4 – Evaluation And Comparative Analysis

The next step is to rigorously evaluate outcomes from the control and active groups. If the active group demonstrates higher accuracy in detecting stress, it suggests that the model provides meaningful support. Conversely, if the control group performs better, the model may require further refinement or alternative approaches. This phase ensures that deployment decisions are evidence-based, minimizing risks of harm or inefficiency.

Phase 5 – Industry Deployment And Scaling

If the model proves effective, it can be gradually introduced into industries such as healthcare, education and workplace wellness programs. In hospitals, it could support psychologists in monitoring patient language. In schools, it could help identify students under stress and in workplaces, it could enhance employee well-being initiatives. However, deployment must come with clear guidelines to prevent over-reliance on the model. Human oversight remains essential as no model can achieve perfect accuracy. The emphasis should be on complementing human expertise rather than replacing it.

Phase 6 – Continuous Monitoring And Ethical Oversight

Finally, long-term success requires ongoing monitoring of the model’s performance, fairness and ethical impact. Regular audits should be conducted to detect biases, ensure privacy protections and evaluate whether the model continues to add value. Feedback loops from professionals and users should inform updates, ensuring that the system evolves responsibly. This phase safeguards against unintended consequences and maintains trust in the technology.

Conclusion:

This project began by introducing the concept of stress and its significance within the medical and mental health domains. We examined both raw text and numerical features from the Dreaddit dataset, including LIWC scores, sentiment values, and DAL indicators. Following a thorough data cleaning process, we conducted exploratory data analysis to uncover meaningful insights, such as correlations between features and stress levels, and the influence of subcategories on stress prevalence in text.

We then evaluated a wide range of models for the classification task, spanning classical machine learning approaches (logistic regression, random forest with boosting and SVM) as well as deep learning architectures (CNN, RNN, LSTM, GRU and bidirectional LSTM). Hybrid neural models were further explored by integrating raw text with numerical features. Each model was assessed using F1-scores and confusion matrices, providing valuable observations about their performance. A cost–benefit analysis was conducted to compare models in terms of accuracy, computational cost and runtime. Ultimately, Hybrid RoBERTa model was selected as the most optimal model, achieving an F1-score of 0.8173 on the Dreaddit test set.

To enhance interpretability, we applied SHAP and LIME to identify the features most influential in stress classification, both locally and globally. We also examined misclassified instances, analyzing common words, phrases, text length and subcategories with higher misclassification rates through word clouds and n-grams. Finally, we discussed the challenges of deploying Hybrid RoBERTa model in real-world settings, considered potential ethical implications, identified sectors where such a model could be applied and outlined a roadmap for future implementation.

Appendix:

Declaration Of AI Usage On Project:

I declare that **I have used AI tools such as ChatGPT and Copilot on some aspects on the project**, including:

- Some code for enhancing the visualizations (formatting, layouts, color designs etc.)
- Sentence structure for several paragraphs to improve on delivery and flow of the project
- Some extensive concepts regarding hyperparameter tuning for transformer models and how to tune them effectively
- Mental health context regarding stress and non-stress indicators (together with some references placed below)
- Certain comments and formatting structure in some lengthy code blocks

References:

1. **Practical guide to SHAP analysis: Explaining supervised machine learning model predictions in drug development:**

   Link: https://pmc.ncbi.nlm.nih.gov/articles/PMC11513550/

   This reference talks about SHAP in-depth and how SHAP analysis can be used in medical trials and research including drug development.

2. **LIME: Interpreting ML Models - A Comprehensive Guide to Local Interpretable Model-agnostic Explanations in Machine Learning**

   Link: https://www.numberanalytics.com/blog/ultimate-guide-lime-mathematics-machine-learning

   This reference explains what LIME is and how LIME analysis can be used in Machine Learning development and optimization processes.

3. **56 Hyperparameter Tuning of a Transformer Network with PyTorch**

   Link: https://sequential-parameter-optimization.github.io/Hyperparameter-Tuning-Cookbook/603_spot_lightning_transformer_hpt.html

   This website contains detailed information regarding how to perform hyperparameter tuning on transformers, some of these concepts applied in the BERT, DistilBERT and RoBERTa models in the project.

4. **Support Vector Machine (SVM) Algorithm**

   Link: https://www.geeksforgeeks.org/machine-learning/support-vector-machine-algorithm/

   This website explains the definitions and knowledge of SVM in a very understandable manner, including the math behind it and the types of SVM.

5. **Bidirectional LSTM in NLP**

   Link: https://www.geeksforgeeks.org/nlp/bidirectional-lstm-in-nlp/

   This website explains the definitions and knowledge of Bidirectional LSTM in a very understandable manner, including the math behind it and how to use Bidirectional LSTM as a deep learning model.