# Submission 1: TFIDF + Random Forest

The model first used was a Random Forest Classifier, which is a powerful ensemble-based machine learning algorithm. I chose this model because it's robust to overfitting on small datasets, especially when features are sparse, as is the case with TF-IDF.Even though it's fast and simple to implement, it still captures complex relationships and non-linear patterns between features without requiring feature scaling. I used this as a baseline for my first submission, and I got a score of 0.33431 in private. It's a good baseline, but it failed to capture sequential or contextual information critical for emotional nuances and still needed improvement.

# Submission 2: LSTM

The second model I used was a stacked LSTM architecture, which I know to be suitable for sequential data like text because it can learn a dense representation of words in a 128-dimensional space and capture semantic relationships between words. LSTM improved the score by approximately 6% compared to TF-IDF + Random Forest, indicating the importance of modeling sequential relationships. However, the model struggled with class imbalance, heavily favoring the majority class (joy). Accuracy stagnated at around 35% after 3 epochs.The learning embeddings from scratch also limited the model's ability to generalize.To me, this was a clear example of underfitting. The final kaggle score was 0.39678.

```
# Build LSTM model
model = Sequential([
    Embedding(input_dim=5000, output_dim=128, input_length=max_seq_length),
    LSTM(128, return_sequences=True),
    Dropout(0.2),
    LSTM(64),
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dense(len(label_encoder.classes_), activation='softmax')
])
print("lstm built")
```

# Submission 3: LSTM + Attention

Building upon the LSTM architecture, the third submission incorporated Bidirectional LSTMs and an Attention mechanism to enhance performance. The Bidirectional LSTMs processed the text in both forward and backward directions, improving contextual understanding. The Attention Mechanism then focused on the most relevant parts of the text sequence, allowing the model to prioritize important words for emotion detection. The dense layers are similar to Submission 2,

with ReLU and softmax layers for classification. I expected the score to rise more, but it only increased by a bit, with a final Kaggle Score of 0.39831.

```
# Build model with Functional API
print("Building model...")
input_layer = Input(shape=(max_seq_length,))
embedding_layer = Embedding(input_dim=10000, output_dim=128,
input_length=max_seq_length, trainable=True)(input_layer)
bi_lstm_layer = Bidirectional(LSTM(128,
return_sequences=True))(embedding_layer)
dropout_layer = Dropout(0.3)(bi_lstm_layer)

# Add Attention Layer
query = dropout_layer  # Query is the output from BiLSTM
value = dropout_layer  # Value is also the BiLSTM output
attention_output = Attention()([query, value])

# Add pooling to reduce sequence dimension
pooled_output = GlobalAveragePooling1D()(attention_output)

# Dense layers for classification
dense_layer = Dense(64, activation='relu')(pooled_output)
dropout_layer_2 = Dropout(0.3)(dense_layer)
output_layer = Dense(len(label_encoder.classes_),
activation='softmax')(dropout_layer_2)

# Create the model
model = Model(inputs=input_layer, outputs=output_layer)
print("Model built.")

# Compile model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

# Insights:

The inclusion of Bidirectional LSTMs and Attention provided a marginal improvement over the standard LSTM model. However, performance remained limited by class imbalance and the absence of pretrained embeddings. The Attention layer helped focus on key parts of the text but did not resolve the fundamental challenges of the dataset. The best course of action would be to

use pre-trained transformer models. Instead of LSTM, a model like BERT or DistilBERT excels in text classification tasks, and it would be possible to use the HuggingFace Transformers library for fine-tuning. Balancing the dataset is also necessary, and it is also possible to explore additional text features like sentiment scores, POS tags, or n-grams to enrich the input data.

# Coding Process:

## Emoji Mapping

Emojis were replaced with descriptive keywords using a predefined dictionary that I made myself (e.g., 😂 → [joy]). This is necessary since the emojis aren't text, so they cannot be processed by the computer. This retained important emotional context present in the tweets while converting emojis into text-compatible features. All other emojis that were not part of the predefined mapping were removed using the emoji library's replace_emoji function.
Later on, I used emoji.demojize to replace emojis with descriptive textual representations (e.g., 😂 → :face_with_tears_of_joy:). This ensured that I no longer had to manually map the emojis, and also considered some less common emojis that I didn't add to my predefined dictionary.

```
emoji_dict = {
    '😂': '[joy]',
    '❤️': '[love]',
    '😍': '[adoration]',
    '😭': '[cry]',
    '❤️': '[care]',
```

```
def preprocess_tweet(text):
    # Replace emojis with descriptive names
    text = emoji.demojize(text)
```

## Text Cleaning and Structure validation

Removed special tags (e.g., <LH>) and URLs using regular expressions to reduce noise.
Trimmed excess spaces and converted all text to lowercase for uniformity.
Ensured that only tweets with valid text entries were processed. Invalid or incomplete entries were skipped. I also removed duplicate tweets in the training set to avoid over-representation of specific samples and prevent potential overfitting.

```
    # Remove special tags and URLs
    text = re.sub(r'<LH>', '', text)
    text = re.sub(r'http\S+|www.\S+', '', text)  # Remove URLs
    # Convert to lowercase and strip extra spaces
    text = text.lower().strip()
    return text
print("clean function done")
```

## Feature Engineering Steps: TFIDF + Random Forest

My feature engineering pipeline focused on transforming the cleaned text into numerical representations suitable for machine learning, which included TF-IDF Vectorization,Train-Test Split, and Label Encoding.

## Feature Engineering Steps: LSTM

For this submission, text features were processed directly for the LSTM model via tokenization, padding, and label encoding.