Group Members:
Rafia Karim (21L-5192)
Ariba Arshad (21L-5381)

Story Generation Model

The objective of this project is to generate an emotionally resonant family-friendly story based on user input using the gpt2SequenceClassifier model and gemini-1.5-flash-latest model.

Emotion Classification

Data Loading:

The dataset we used had three files i.e. training.csv, test.csv and validation.csv. Using pd.read_csv, we read the files in train_df, test_df and val_df. They all had two columns (text and label). train_df had 16000 rows, while test_df and val_df both had 2000 rows.

```
print(train_df.shape)
print(test_df.shape)
print(val_df.shape)
```

```
(16000, 2)
(2000, 2)
(2000, 2)
```

In train_df[‘label’], there were six unique values (emotions).

```
unique_emotions = train_df['label'].value_counts()

# Display the unique emotions and their counts
print(unique_emotions)
```

```
label
1    5362
0    4666
3    2159
4    1937
2    1304
5     572
Name: count, dtype: int64
```

According to the dataset description:

sadness (0), joy (1), love (2), anger (3), fear (4), surprise (5). So we mapped them for simplicity.

```
emotion_mapping = {
    0: 'sadness',
    1: 'joy',
    2: 'love',
    3: 'anger',
    4: 'fear',
    5: 'surprise'
}
```

Data Preprocessing:

- **Remove NULL values if present**

  There were no NULL values in any dataframe.

```
print("Null values in train_df:")
print(train_df.isnull().sum())

print("\nNull values in test_df:")
print(test_df.isnull().sum())

print("\nNull values in val_df:")
print(val_df.isnull().sum())
```

```
Null values in train_df:
text     0
label    0
dtype: int64

Null values in test_df:
text     0
label    0
dtype: int64

Null values in val_df:
text     0
label    0
dtype: int64
```

- **Remove duplicates if present**

  Train_df had one duplicate so we removed it.

```
print("Duplicate rows in train_df:", train_df.duplicated().sum())
print("Duplicate rows in test_df:", test_df.duplicated().sum())
print("Duplicate rows in val_df:", val_df.duplicated().sum())
```

```
Duplicate rows in train_df: 1
Duplicate rows in test_df: 0
Duplicate rows in val_df: 0
```

```
train_df = train_df.drop_duplicates()
```

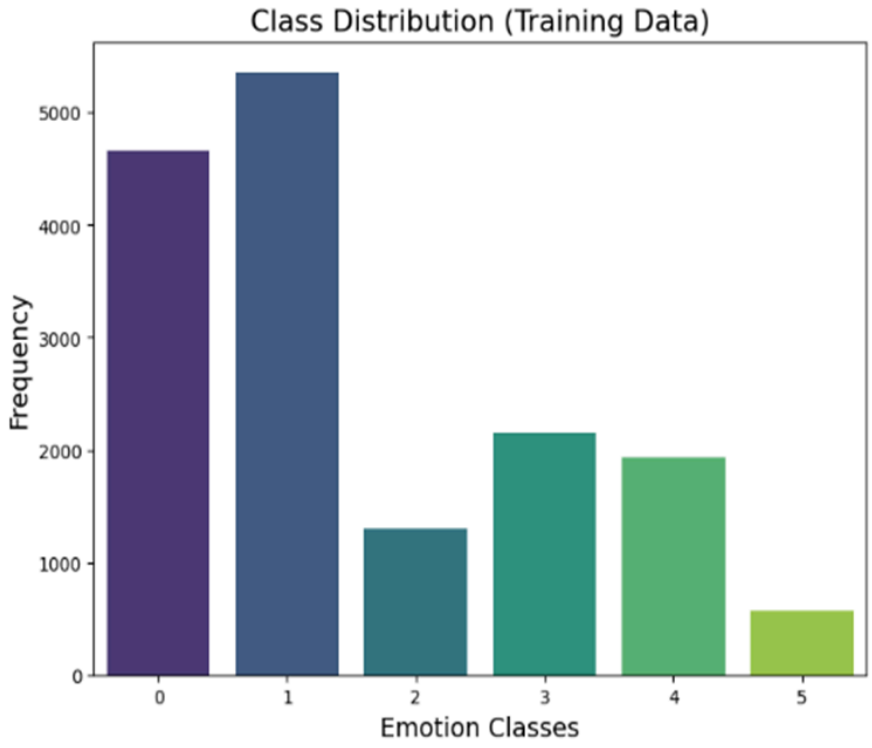
- **Remove special characters and numbers**

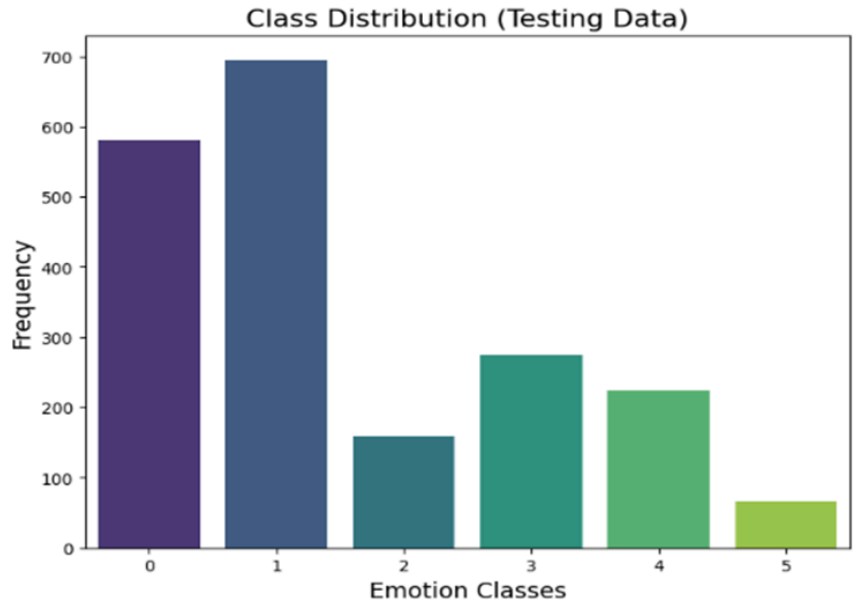
  To clean the data, we removed special characters and numbers from text.

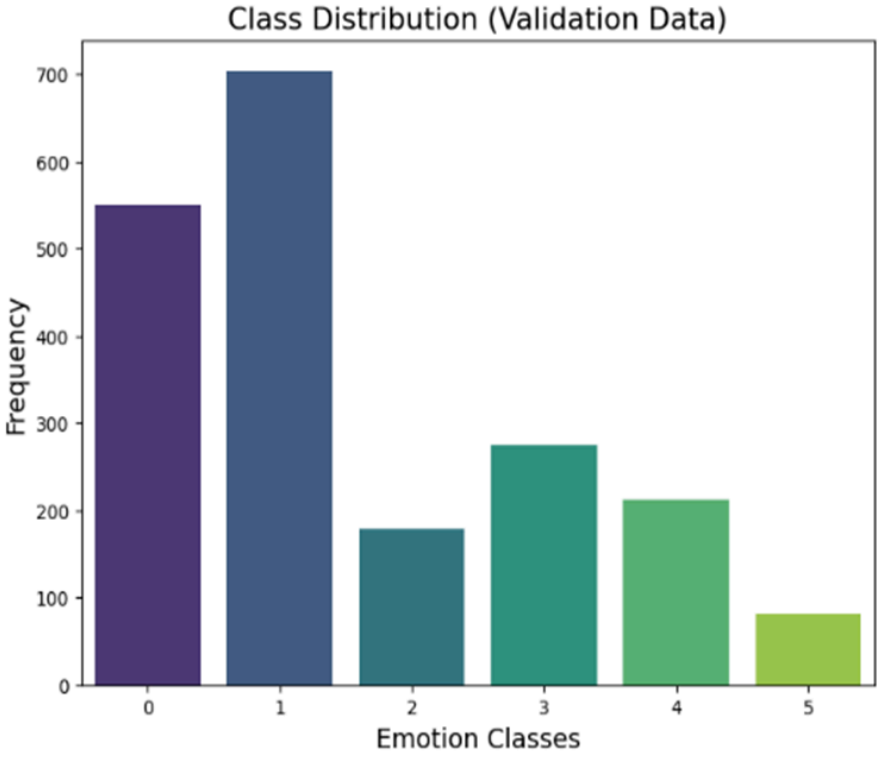
```python
import re

def clean_text(text):
    text = re.sub(r'[^A-Za-z\s]', '', text)  # Keep only letters and spaces
    text = re.sub(r'\s+', ' ', text).strip()  # Remove extra spaces
    return text

train_df['text'] = train_df['text'].apply(clean_text)
test_df['text'] = test_df['text'].apply(clean_text)
val_df['text'] = val_df['text'].apply(clean_text)
```

Visualization:

- **Bar charts**

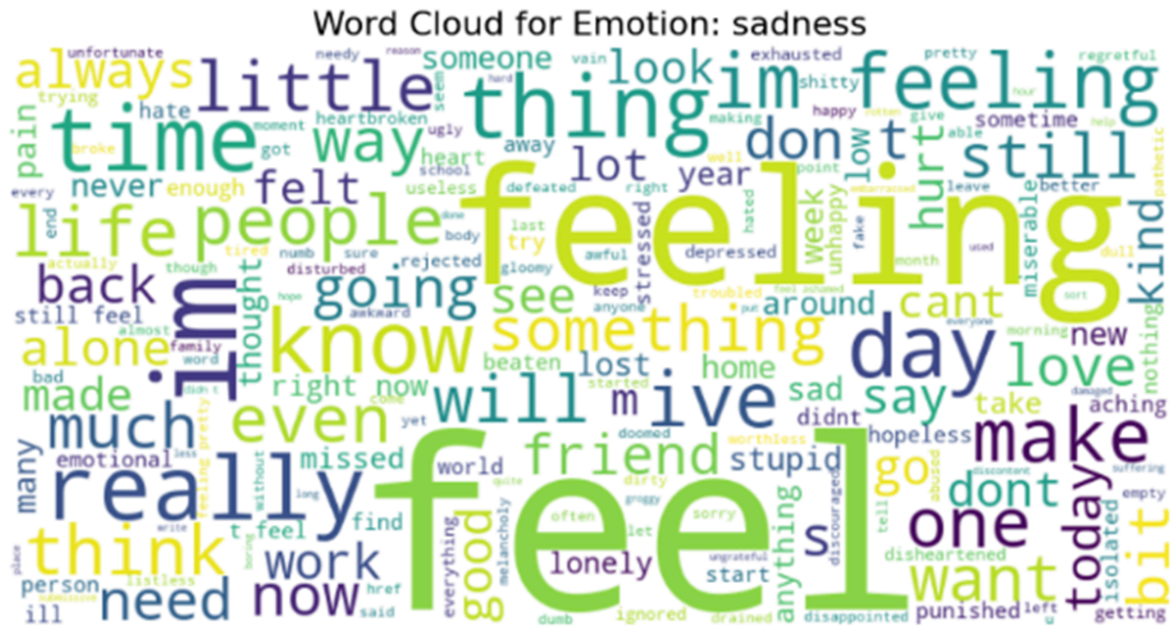


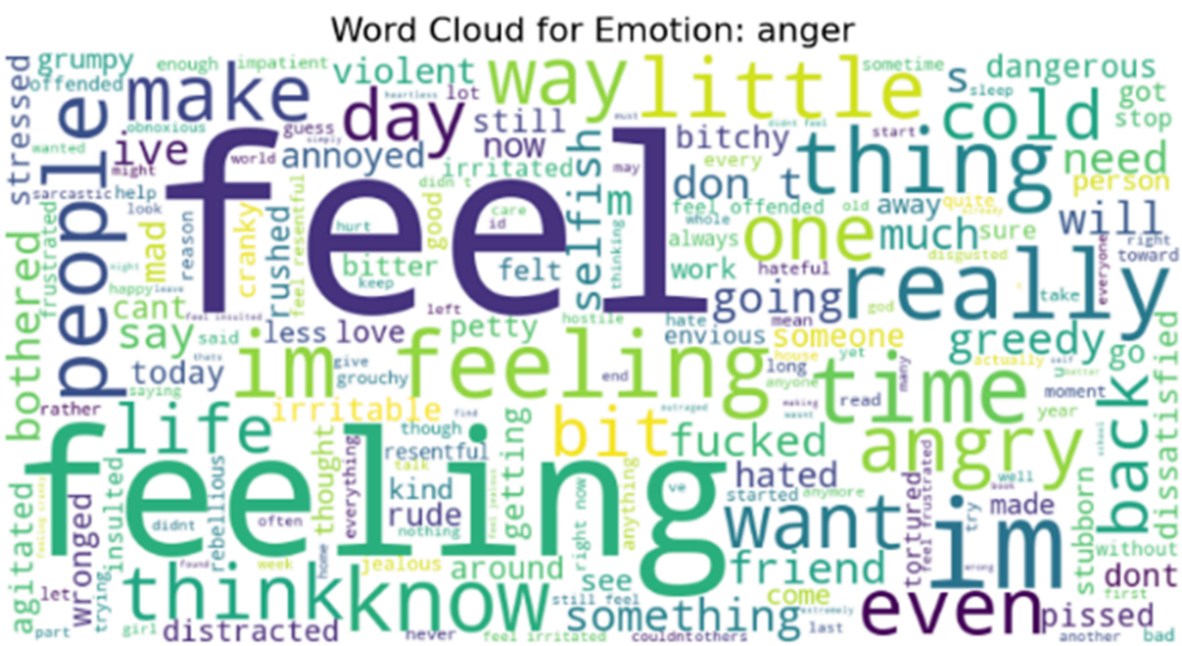




To balance classes, we experimented with SMOTE but the model accuracy did not improve so to decrease training time we did not apply SMOTE in our final implementation.

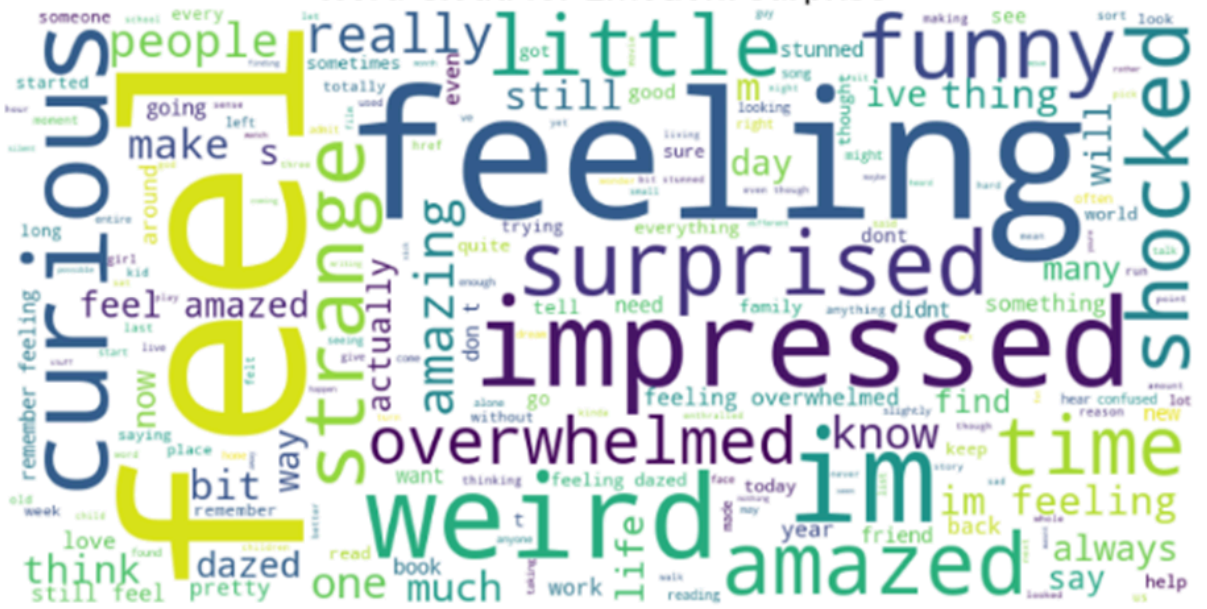
- **Wordclouds**


Word Cloud for Emotion: sadness
feel
feeling
im feeling
thing
time
little
always
know
im
really
day
something
people
life
going
ive
will
make
one
want
think
need
now
friend
even
love
alone
back
much
still
way
look
don t
lot
year
felt
see
cant
say
good
today
bit
work
lonely
stupid
hopeless
emotional
missed
lost
home
sad
kind
hurt
miserable
unhappy
week
depressed
stressed
thought
anything
heartbroken
someone
pain
never
made
person
start
punished

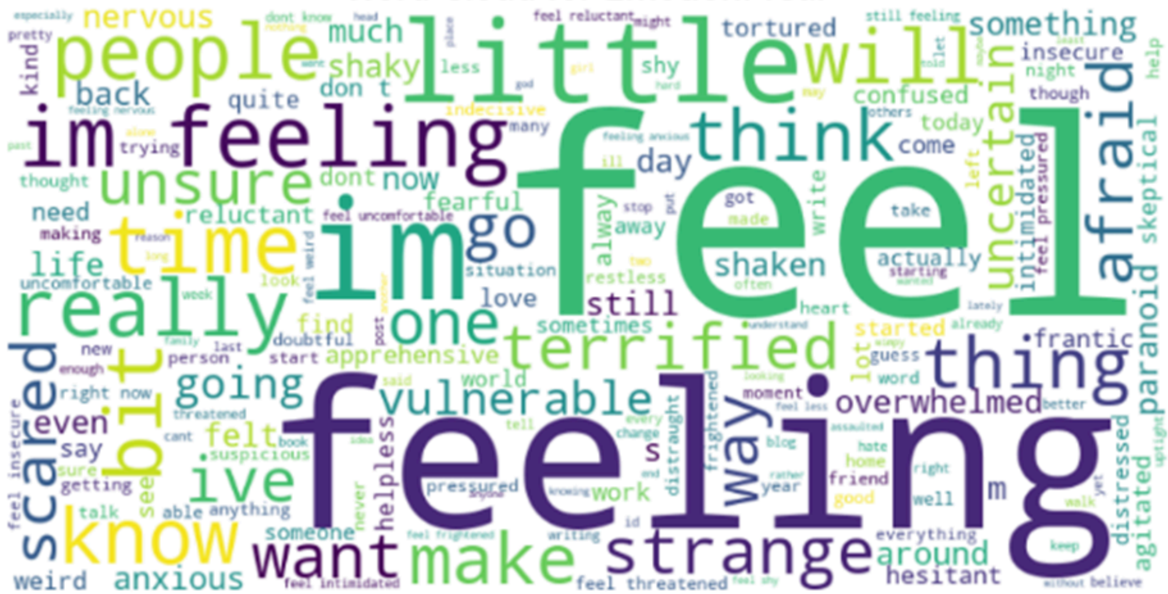

Word Cloud for Emotion: anger
feel
feeling
im feeling
really
thing
people
way
little
make
day
cold
time
life
think
know
want
im
even
angry
back
bit
fucked
friend
something
greedy
one
much
going
need
violent
dangerous
annoyed
irritated
bitchy
selfish
bothered
stressed
dissatisfied
irritable
rude
petty
bitter
hated
pissed
distracted
agitated
wronged
insulted
rebellious
grumpy
cranky
rushed
mad
hateful
envious
tortured
stubborn
jealous

Word Cloud for Emotion: love

Word Cloud for Emotion: surprise



Word Cloud for Emotion: fear



Word Cloud for Emotion: joy

Modeling

- First we converted dataframes into hugging face datasets.

```
# Convert DataFrames to Hugging Face Dataset format
train_dataset = Dataset.from_pandas(train_df)
test_dataset = Dataset.from_pandas(test_df)
val_dataset = Dataset.from_pandas(val_df)
```

- We loaded the gpt2 tokenizer and set the padding token to end-of-sentence token to ensure that all tokenized sentences have the same length.

```
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token #padding
```

- Next, we defined tokenize_function in which tokenizer takes the text data and if its length is greater than 128, truncates it else if its length is less than 128, it applies padding. The function return tokenized sequences.

```
def tokenize_function(examples):
    return tokenizer(examples['text'], truncation=True, padding="max_length", max_length=128)
```

```
train_dataset = train_dataset.map(tokenize_function, batched=True)
test_dataset = test_dataset.map(tokenize_function, batched=True)
val_dataset = val_dataset.map(tokenize_function, batched=True)
```

- Since we tokenized the text data so we no longer need the plaintext and dropped the column.

```
train_dataset = train_dataset.remove_columns(["text"])
test_dataset = test_dataset.remove_columns(["text"])
val_dataset = val_dataset.remove_columns(["text"])
```

- After it we converted datasets to tensors to make them compatible with gpt2 sequence classifier.

```
train_dataset.set_format("torch")
test_dataset.set_format("torch")
val_dataset.set_format("torch")
```

- Next we loaded the gpt2SequenceClassifier model and resized its embedding layer according to new tokens/vocabulary learned by tokenizer. Also we set its padding token to that of the tokenizer.

```
model = GPT2ForSequenceClassification.from_pretrained("gpt2", num_labels=len(emotion_mapping))
# Set the default embedding layer to accomodate new tokens/vocabulary
model.resize_token_embeddings(len(tokenizer))
```

```
model.config.pad_token_id = tokenizer.pad_token_id
```

- We defined training arguments for the model fine-tuning. We defined weight_decay to penalize large weights.

```
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    logging_dir="./logs",
    learning_rate=5e-5,
    per_device_train_batch_size=4,
    num_train_epochs=2,
    weight_decay=0.01, #to penalize large weights
    report_to="none",  # Disable W&B logging
    logging_steps=50
)
```

- Then we trained the model providing training and validation datasets.

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
)
```

```
trainer.train()
```

Results and Evaluation:

- We tested the trained model on test dataset. The accuracy of test data is 92.85%.

```
test_results = trainer.evaluate(test_dataset)

print(test_results)
```

```
{'eval_loss': 0.17025277018547058, 'eval_accuracy': 0.9285, 'eval_runtime': 14.1734, 'eval_samples_per_second': 141.109, 'eval_
steps_per_second': 8.819, 'epoch': 2.0}
```

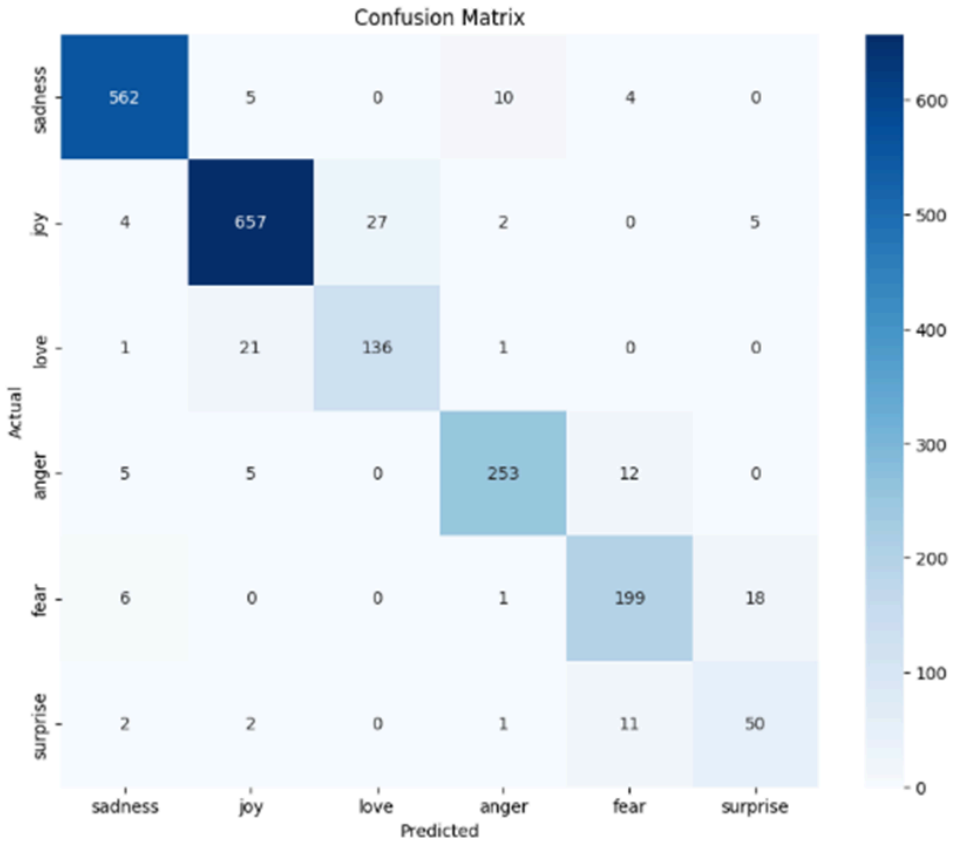
- Evaluation metrics

```
Evaluation Metrics:
              precision    recall  f1-score   support

     sadness       0.97      0.97      0.97       581
         joy       0.95      0.95      0.95       695
        love       0.83      0.86      0.84       159
       anger       0.94      0.92      0.93       275
        fear       0.88      0.89      0.88       224
    surprise       0.68      0.76      0.72        66

    accuracy                           0.93      2000
   macro avg       0.88      0.89      0.88      2000
weighted avg       0.93      0.93      0.93      2000

Accuracy: 0.93
```

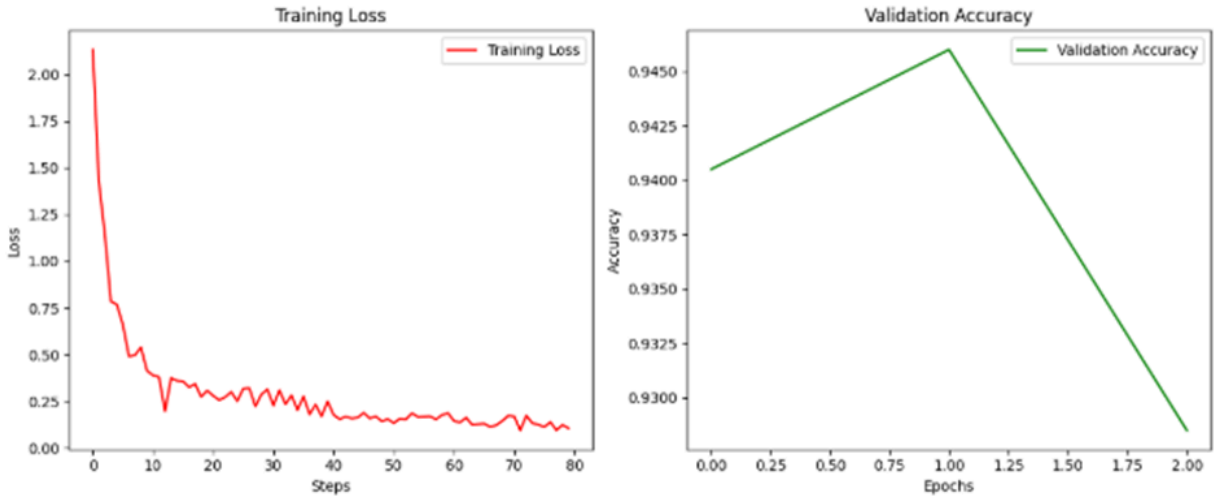
Here,

- support shows the number of instances of each emotion in the test dataset.
- precision = TP/(TP+FP)
- recall = TP/(TP+FN)
- f1-score = (2*precision*recall) / (precision+recall)

- Confusion matrix



- Training loss and Accuracy



Story Generation

The story description input is checked to ensure that it aligns with specific guidelines and the story is generated with specific parameters for structure, content and length.

Pre-trained gemini-1.5-flash-latest model is used

- to check if user input matches specific criteria
- to generate the story text based on the user input

```
model = genai.GenerativeModel('gemini-1.5-flash-latest')
```

The process starts by prompting the user to enter the story description.

```
user_input = input('Enter story description: ')
```

The user input is checked to ensure that it contains only one human character. This is done by prompting the gemini-1.5-flash-latest model to return the number of human characters in the prompt.

```
prompt = 'Return number of the humans in the prompt. Just return number. '
prompt += user_input
response = model.generate_content(prompt).text.lower().strip()
```

If the prompt contains only one human character then it is checked if the prompt can be used to generate a story. This is also done by prompting the gemini-1.5-flash-latest model to return ‘acceptable’ if the user input is descriptive enough to be used for story generation.

```
prompt = "Please return 'acceptable' if the user provided prompt is decriptive enough to generate a meaningful story."
prompt += user_input
response = model.generate_content(prompt).text.lower().strip()
```

If any of these two conditions is not satisfied then the user is prompted to enter the story description again.

```
user_input = input('Enter story description again (it should be descriptive and have one character only): ')
```

When both conditions are satisfied then the emotion of the user input is classified and prompt engineering is used to prepare the prompt which will be sent to the gemini-1.5-flash-latest model to generate a story.
The emotion of the user input is determined using the fine-tuned gpt2SequenceClassifier model.

```
predicted_emotion = predict_emotion(user_input)
```

The prompt engineering includes:

- Family-friendly story
- Approximately 3000 words length
- In the form of paragraphs
- Have clear beginning, middle and end
- Emotionally resonant
- As there is only one main character so the dialogues will be the thoughts of the character
- Adding user story description input

```
word_count = '3000'
prompt = 'Generate a family friendly story. '
prompt += 'Use engaging paragraphs. '
prompt += 'The story should have a clear beginning, middle, and end. '
prompt += 'The character should not be carrying any stuff like bag. '
prompt += 'The main character should be alone, with no significant mention of any other human character in the story. '
prompt += 'All dialogue should be only the thoughts of the main character, in quotation marks. '
prompt += f'The length of story should be approximately {word_count} words. '
prompt += f'The story should have {predicted_emotion} emotion throughout to ensure emotionally resonant story. '
prompt += 'The description of the story is: '
prompt += user_input
```

Lastly, the story is generated and displayed.

```
response = model.generate_content(prompt)

print(response.text)
```

The prompt engineering has been used to ensure the story generation adheres to specific guidelines and user requirements. The prompt engineering is used at three occasions:

- Determining number of human characters in the user input
- Determining if meaningful story can be generated using the user provided story description
- Crafting the prompt before sending it to the model for story generation

Conclusion

The gpt2SequenceClassifier model and gemini-1.5-flash-latest model have been used to generate an emotionally resonant story. The gpt2SequenceClassifier model has been used to classify the emotion of user input and gemini-1.5-flash-latest model has been used to generate story after checking specific constraints. This integration ensures that the generated story aligns with the user's input and adheres to the defined constraints and delivers an emotionally resonant story.