M508C

Big Data Analytics (WS0925)

Individual Final Project

Student Name: Anh Tuan Ta
Student ID: GH1046139.
Student Email: anh.ta@gisma-student.com
LinkedIn: https://www.linkedin.com/in/ta-anh-tuan-ai-engineer

MSc of DATA SCIENCE, AI AND DIGITAL BUSINESS

Emotion Classification on Tweets using Natural Language Processing

Problem Statement

Business Context

Social media platforms such as Twitter generate large volumes of short, informal text reflecting users' emotions, opinions, and reactions to real-world events. These emotional signals are valuable for applications such as customer feedback analysis, brand monitoring, and public sentiment tracking.

However, manual analysis of emotional content at scale is infeasible. Tweets are typically noisy, short, and informal, often containing slang, emojis, hashtags, abbreviations, and uneven emotion distributions. These characteristics make automated emotion classification a challenging NLP task.

An effective automated emotion classification system enables organizations to systematically extract emotional insights from social media streams and support:

- Customer experience analysis
- Marketing and brand sentiment monitoring
- Social trend and opinion analysis

Objective

The objective of this project is to design and evaluate a supervised NLP pipeline that classifies the **emotion expressed in a tweet**.

Specifically, the system aims to:

- Take raw tweet text as input
- Predict one of six emotion labels:
  sadness, joy, love, anger, fear, surprise
- Apply practical preprocessing suitable for social media text
- Implement and compare **three NLP modeling approaches** covered in the course:
  - **TF-IDF + Logistic Regression**
  - **Word2Vec embeddings + LSTM**
  - **Transformer-based model (BERT fine-tuning)**
- Mitigate class imbalance using **class-weighted training**
- Evaluate models using both overall and class-balanced metrics
- Analyze strengths and weaknesses of each modeling approach

Dataset Description

Dataset

- **Source:** Public Twitter Emotion Dataset (Kaggle)
- **Total samples:** Approximately 20,000 tweets
- **Data splits:**
  - `training.csv`
  - `validation.csv`
  - `test.csv`
- **Input:** Raw tweet text
- **Target:** One of six emotion categories

Class Distribution

The dataset is moderately imbalanced, with dominant emotions such as **joy** and **sadness**, and minority classes such as **love** and **surprise**.

Approximate distribution:

- Joy ≈ 33%
- Sadness ≈ 29%
- Anger ≈ 13%
- Fear ≈ 12%
- Love ≈ 8%
- Surprise ≈ 4%

This imbalance motivates the use of **macro-averaged metrics** and **class-weighted training**, especially for neural models.

Modeling Pipeline & Approach

Exploratory Analysis

The project begins with exploratory data analysis to understand:

- Emotion class distribution
- Tweet length statistics
- Vocabulary characteristics

Text Preprocessing

Preprocessing steps implemented in the notebook include:

- Lowercasing
- Tokenization
- Removal of unnecessary punctuation and symbols
- Lemmatization
- Padding and truncation for sequence-based models

No explicit normalization of elongated words or repeated characters is applied.

Model Architectures

Three modeling pipelines are implemented and evaluated:

1. **TF-IDF + Logistic Regression**
    - Sparse bag-of-words representation
    - Strong linear baseline
    - Interpretable and computationally efficient
2. **Word2Vec + Recurrent Neural Network**
    - Pre-trained Word2Vec embeddings
    - Sequence modeling using LSTM
    - Class-weighted loss to address imbalance
3. **Transformer-based Model (BERT)**
    - Fine-tuned pre-trained BERT model
    - Context-aware token representations
    - End-to-end supervised fine-tuning

Evaluation Strategy

Given the imbalanced multi-class setting, model evaluation focuses on both overall performance and per-class behavior.

Primary Metrics

- **Macro F1-score**
  Ensures equal importance for minority emotion classes.
- **Per-class Precision, Recall, and F1-score**
  Highlights model behavior on difficult emotions such as *love* and *surprise*.

Secondary Metrics

- Overall accuracy
- Weighted F1-score
- Confusion matrices for error analysis
- Training and validation curves (for neural models)

Expected Outcomes

- A complete end-to-end emotion classification pipeline for tweets
- Empirical comparison between classical ML, neural sequence models, and transformers
- Insights into how model complexity and contextual representation affect emotion detection
- Identification of limitations related to data imbalance and short-text semantics

This project demonstrates how increasingly context-aware models improve emotion classification performance in noisy social-media environments.

1. Imports and Config

In [1]:
```python
import os
import random
from collections import Counter

import numpy as np
import pandas as pd

import re
from itertools import chain
import emoji

import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud

from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    confusion_matrix,
    classification_report,
)
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.utils.class_weight import compute_class_weight

from imblearn.over_sampling import RandomOverSampler

import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

from gensim.models import Word2Vec
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras import layers, models, regularizers

from transformers import TrainingArguments

from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    Trainer,
    TrainingArguments,
)
import torch
from torch.utils.data import Dataset
```

```
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/tqdm/aut
o.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywi
dgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [2]:

```
# Reproducibility
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

In [3]:

```
# NLTK downloads
nltk.download("punkt")
nltk.download("stopwords")
nltk.download("wordnet")
```

```
[nltk_data] Downloading package punkt to /Users/taanhtuan/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/taanhtuan/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     /Users/taanhtuan/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

Out[3]:

```
True
```

In [4]:

```
stop_words = set(stopwords.words("english"))
lemmatizer = WordNetLemmatizer()
```

In [5]:
```
EMOTION_DICT = {
    0: "sadness",
    1: "joy",
    2: "love",
    3: "anger",
    4: "fear",
    5: "surprise"
}
```

2. Load Data

In [6]:
```
DATA_DIR = "data/Emotion6"
```

In [7]:
```
TRAIN_PATH = os.path.join(DATA_DIR, "training.csv")
DEV_PATH   = os.path.join(DATA_DIR, "validation.csv")
TEST_PATH  = os.path.join(DATA_DIR, "test.csv")
```

In [8]:
```
train_df = pd.read_csv(TRAIN_PATH)
dev_df   = pd.read_csv(DEV_PATH)
test_df  = pd.read_csv(TEST_PATH)
```

In [9]:
```
print("Train shape:", train_df.shape)
```
```
Train shape: (16000, 2)
```

In [10]:
```
print("Dev shape  :", dev_df.shape)
```
```
Dev shape  : (2000, 2)
```

In [11]:
```
print("Test shape :", test_df.shape)
```
```
Test shape : (2000, 2)
```

In [12]:
```
display(train_df.head())
```

	text	label
0	i didnt feel humiliated	0
1	i can go from feeling so hopeless to so damned...	0
2	im grabbing a minute to post i feel greedy wrong	3
3	i am ever feeling nostalgic about the fireplac...	2
4	i am feeling grouchy	3

In [13]:
```
display(dev_df.head())
```

	text	label
0	im feeling quite sad and sorry for myself but ...	0
1	i feel like i am still looking at a blank canv...	0
2	i feel like a faithful servant	2
3	i am just feeling cranky and blue	3
4	i can have for a treat or if i am feeling festive	1

In [14]:
```
display(test_df.head())
```

	text	label
0	im feeling rather rotten so im not very ambiti...	0
1	im updating my blog because i feel shitty	0
2	i never make her separate from me because i do...	0
3	i left with my bouquet of red and yellow tulip...	1
4	i was feeling a little vain when i did this one	0

3. Data Preprocessing + EDA

3.1 Text Cleaning + Encoding

Remove empty text and label

In [15]:
```
train_df = train_df.dropna(subset=["text", "label"])
dev_df   = dev_df.dropna(subset=["text", "label"])
test_df  = test_df.dropna(subset=["text", "label"])
```

Label Encoding

In [16]:
```
label_encoder = LabelEncoder()
label_encoder.fit(train_df["label"])
```

Out[16]:
```
▾ LabelEncoder
▶ Parameters
```

In [17]:
```
train_df["label_id"] = label_encoder.transform(train_df["label"])
dev_df["label_id"]   = label_encoder.transform(dev_df["label"])
test_df["label_id"]  = label_encoder.transform(test_df["label"])
```

In [18]:
```
NUM_CLASSES = len(label_encoder.classes_)
NUM_CLASSES
```

Out[18]:
```
6
```

In [19]:
```python
print("Classes:", list(label_encoder.classes_))
```

```
Classes: [np.int64(0), np.int64(1), np.int64(2), np.int64(3), np.int64(4),
np.int64(5)]
```

Detect emojis & hashtags

In [20]:
```python
def extract_emojis(text):
    text = str(text)
    return ''.join(ch for ch in text if ch in emoji.EMOJI_DATA)

def extract_hashtags(text):
    text = str(text)
    return re.findall(r'#\w+', text)

# add columns to train_df

train_df["emojis"] = train_df["text"].astype(str).apply(extract_emojis)
train_df["num_emojis"] = train_df["emojis"].str.len()

train_df["hashtags"] = train_df["text"].astype(str).apply(extract_hashtag
train_df["num_hashtags"] = train_df["hashtags"].str.len()
```

In [21]:
```python
print("Emoji / Hashtag Usage in Training Set")

print("Tweets with ≥1 emoji    :", (train_df["num_emojis"] > 0).mean() *
print("Tweets with ≥1 hashtag  :", (train_df["num_hashtags"] > 0).mean()

print("\nAverage emojis per tweet  :", train_df["num_emojis"].mean())
print("Average hashtags per tweet:", train_df["num_hashtags"].mean())
```

```
Emoji / Hashtag Usage in Training Set
Tweets with ≥1 emoji    : 0.0 %
Tweets with ≥1 hashtag  : 0.0 %

Average emojis per tweet  : 0.0
Average hashtags per tweet: 0.0
```

Quick Look: Emoji & Hashtag Presence

The dataset contains **no detectable emojis or hashtags**:

- **Tweets with ≥ 1 emoji:** 0.0%
- **Tweets with ≥ 1 hashtag:** 0.0%
- **Average emojis per tweet:** 0.0
- **Average hashtags per tweet:** 0.0

This confirms that emojis and hashtags **do not appear** in the dataset, so keeping/removing them **has no impact** on preprocessing or model performance.

Create Raw + Cleaned Text Lists

In [22]:
```
def clean_tokens(text):
    text = str(text).lower()
    tokens = nltk.word_tokenize(text)
    tokens = [t for t in tokens if t.isalpha()]
    tokens = [t for t in tokens if t not in stop_words]
    tokens = [lemmatizer.lemmatize(t) for t in tokens]
    return tokens
```

In [23]:
```
train_df["clean_tokens"] = train_df["text"].astype(str).apply(clean_token
train_df["clean_text"] = train_df["clean_tokens"].apply(lambda x: " ".joi
```

In [24]:
```
print("Display few examples:")
for i in range(3):
    print(f"RAW   : {train_df.loc[i, 'text']}")
    print(f"CLEAN : {train_df.loc[i, 'clean_text']}")
    print("---")
```

```
Display few examples:
RAW   : i didnt feel humiliated
CLEAN : didnt feel humiliated
---
RAW   : i can go from feeling so hopeless to so damned hopeful just from b
eing around someone who cares and is awake
CLEAN : go feeling hopeless damned hopeful around someone care awake
---
RAW   : im grabbing a minute to post i feel greedy wrong
CLEAN : im grabbing minute post feel greedy wrong
---
```

In [25]:
```
dev_df["clean_tokens"] = dev_df["text"].astype(str).apply(clean_tokens)
dev_df["clean_text"] = dev_df["clean_tokens"].apply(lambda x: " ".join(x)
```

In [26]:
```
print("Display few examples:")
for i in range(3):
    print(f"RAW   : {dev_df.loc[i, 'text']}")
    print(f"CLEAN : {dev_df.loc[i, 'clean_text']}")
    print("---")
```

```
Display few examples:
RAW   : im feeling quite sad and sorry for myself but ill snap out of it s
oon
CLEAN : im feeling quite sad sorry ill snap soon
---
RAW   : i feel like i am still looking at a blank canvas blank pieces of p
aper
CLEAN : feel like still looking blank canvas blank piece paper
---
RAW   : i feel like a faithful servant
CLEAN : feel like faithful servant
---
```

In [27]:
```
test_df["clean_tokens"] = test_df["text"].astype(str).apply(clean_tokens)
test_df["clean_text"] = test_df["clean_tokens"].apply(lambda x: " ".join(
```

In [28]:
```
print("Display few examples:")
for i in range(3):
    print(f"RAW   : {test_df.loc[i, 'text']}")
    print(f"CLEAN : {test_df.loc[i, 'clean_text']}")
    print("---")
```

```
Display few examples:
RAW   : im feeling rather rotten so im not very ambitious right now
CLEAN : im feeling rather rotten im ambitious right
---
RAW   : im updating my blog because i feel shitty
CLEAN : im updating blog feel shitty
---
RAW   : i never make her separate from me because i don t ever want her to
feel like i m ashamed with her
CLEAN : never make separate ever want feel like ashamed
---
```

3.2 Exploratory Data Analysis (EDA)

Percentages of each label

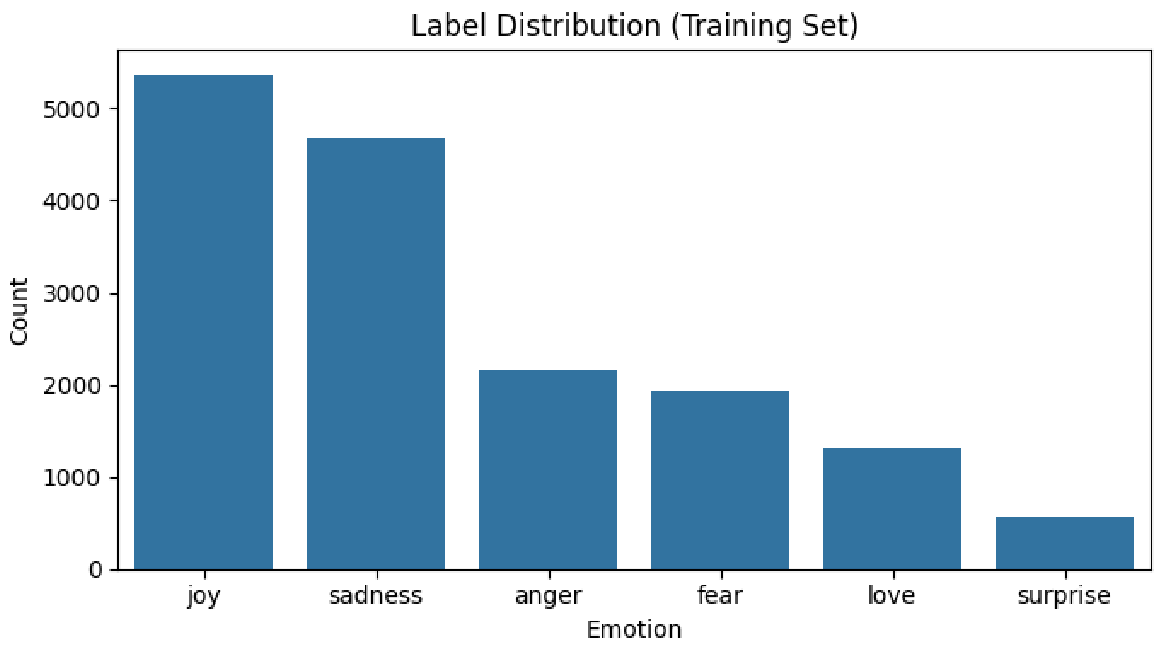
In [29]:
```
for lbl, count in train_df["label"].value_counts().items():
    pct = count / len(train_df) * 100
    print(f"Label {lbl}: {count:,} samples ({pct:.2f}%)")
```

```
Label 1: 5,362 samples (33.51%)
Label 0: 4,666 samples (29.16%)
Label 3: 2,159 samples (13.49%)
Label 4: 1,937 samples (12.11%)
Label 2: 1,304 samples (8.15%)
Label 5: 572 samples (3.57%)
```

Label Distribution Visualization

In [30]:
```
# Order by frequency of numeric labels
order_labels = train_df["label"].value_counts().index

plt.figure(figsize=(7,4))
ax = sns.countplot(
    x="label",
    data=train_df,
    order=order_labels
)

# Replace tick labels (0..5) with emotion names
ax.set_xticklabels([EMOTION_DICT[i] for i in order_labels])

plt.title("Label Distribution (Training Set)")
plt.xlabel("Emotion")
plt.ylabel("Count")
plt.tight_layout()
plt.show()
```

```
/var/folders/6x/h489kd6s4lxbdnxrhw8mvt140000gn/T/ipykernel_52202/98844256
2.py:12: UserWarning: set_ticklabels() should only be used with a fixed nu
mber of ticks, i.e. after set_ticks() or using a FixedLocator.
  ax.set_xticklabels([EMOTION_DICT[i] for i in order_labels])
```



Interpretation

The training dataset shows a clear **class imbalance**:

- **Joy (label 1)** and **Sadness (label 0)** are the most common classes.
- **Anger (3)** and **Fear (4)** appear moderately often.
- **Love (2)** and especially **Surprise (5)** are **minority classes**.

This imbalance is important because it can lead to:

- Lower recall and F1-scores for rare classes (e.g., *love*, *surprise*)
- Bias toward predicting majority classes
- Unstable training for neural models

To address this, the pipeline applies **oversampling** and **class weighting**, ensuring that all models treat minority emotions more fairly during training.

Average Tweet Length (in words) per Emotion

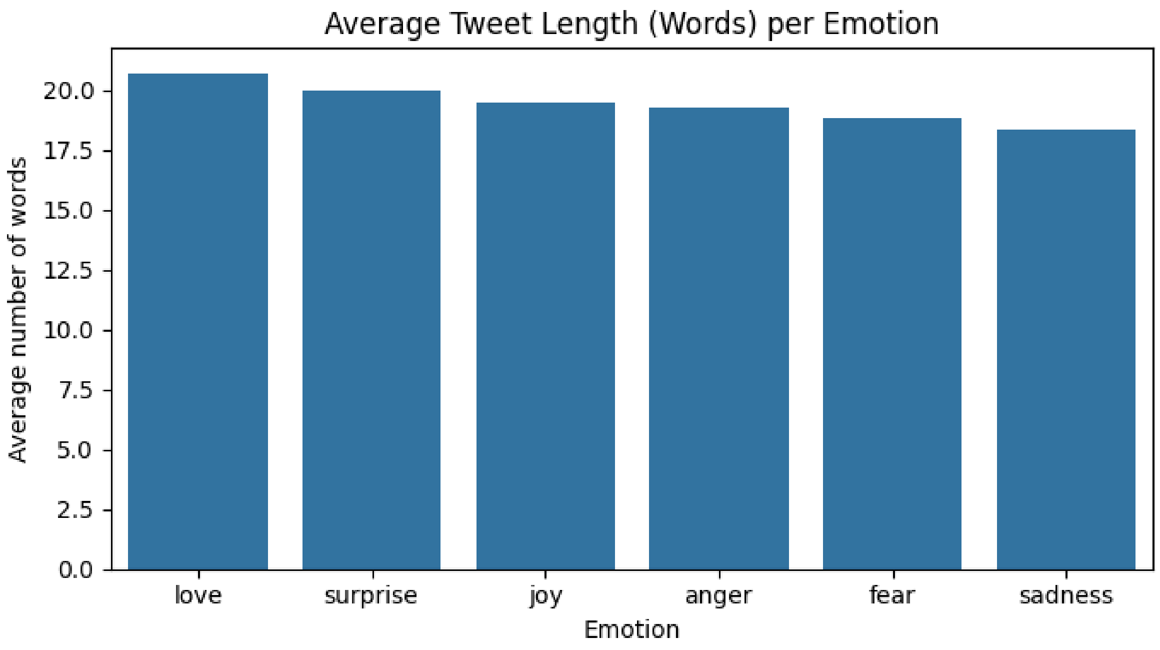
In [31]:
```
# Add num_words feature
for df_part in (train_df, dev_df, test_df):
    df_part["num_words"] = df_part["text"].astype(str).apply(lambda x: le
```

In [32]:
```
# Compute average length per label
avg_len = (
    train_df
    .groupby("label")["num_words"]
    .mean()
    .reset_index()
    .rename(columns={"num_words": "avg_num_words"})
)
```

In [33]:
```
# Map numeric label -> emotion name
avg_len["emotion"] = avg_len["label"].map(EMOTION_DICT)
```

```
# Sort by average length (optional)
avg_len = avg_len.sort_values("avg_num_words", ascending=False)
```

In [34]:
```
plt.figure(figsize=(7, 4))
sns.barplot(data=avg_len, x="emotion", y="avg_num_words")
plt.title("Average Tweet Length (Words) per Emotion")
plt.xlabel("Emotion")
plt.ylabel("Average number of words")
plt.tight_layout()
plt.show()
```



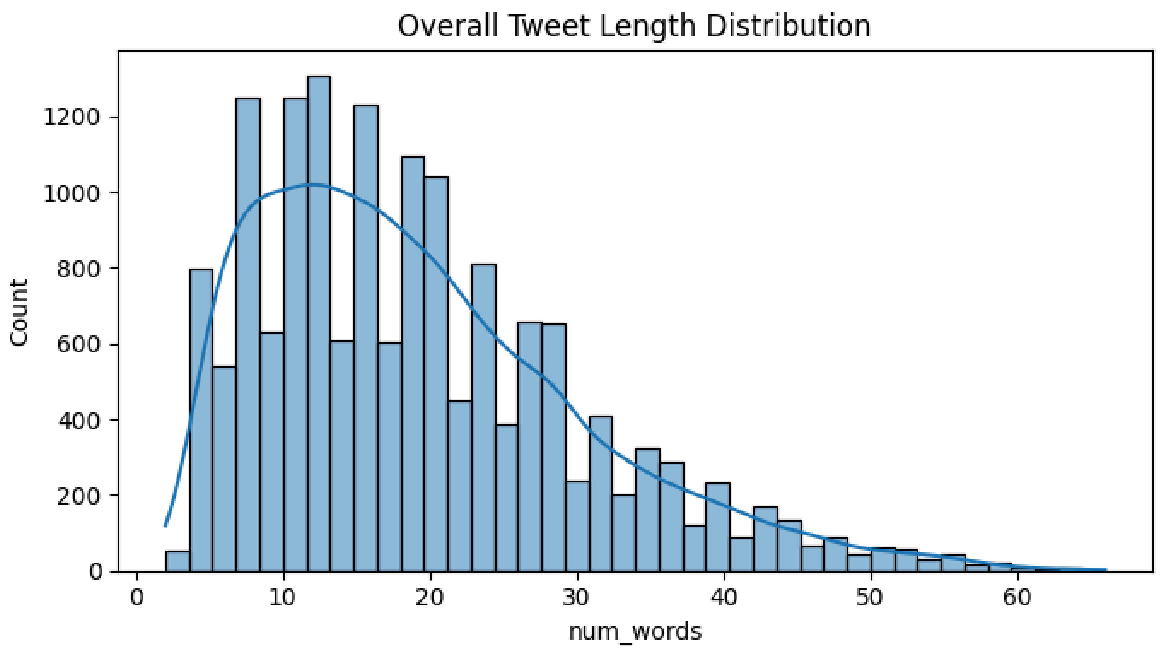
Quick Look – Average Tweet Length per Emotion

- **Love** tweets are the longest (~21 words), often containing more detailed emotional expression.
- **Surprise** and **Joy** follow (~19–20 words), usually describing events or reactions.
- **Anger** and **Fear** are slightly shorter (~19 words), often more direct.
- **Sadness** tweets are the shortest (~18 words), typically brief statements of feeling.

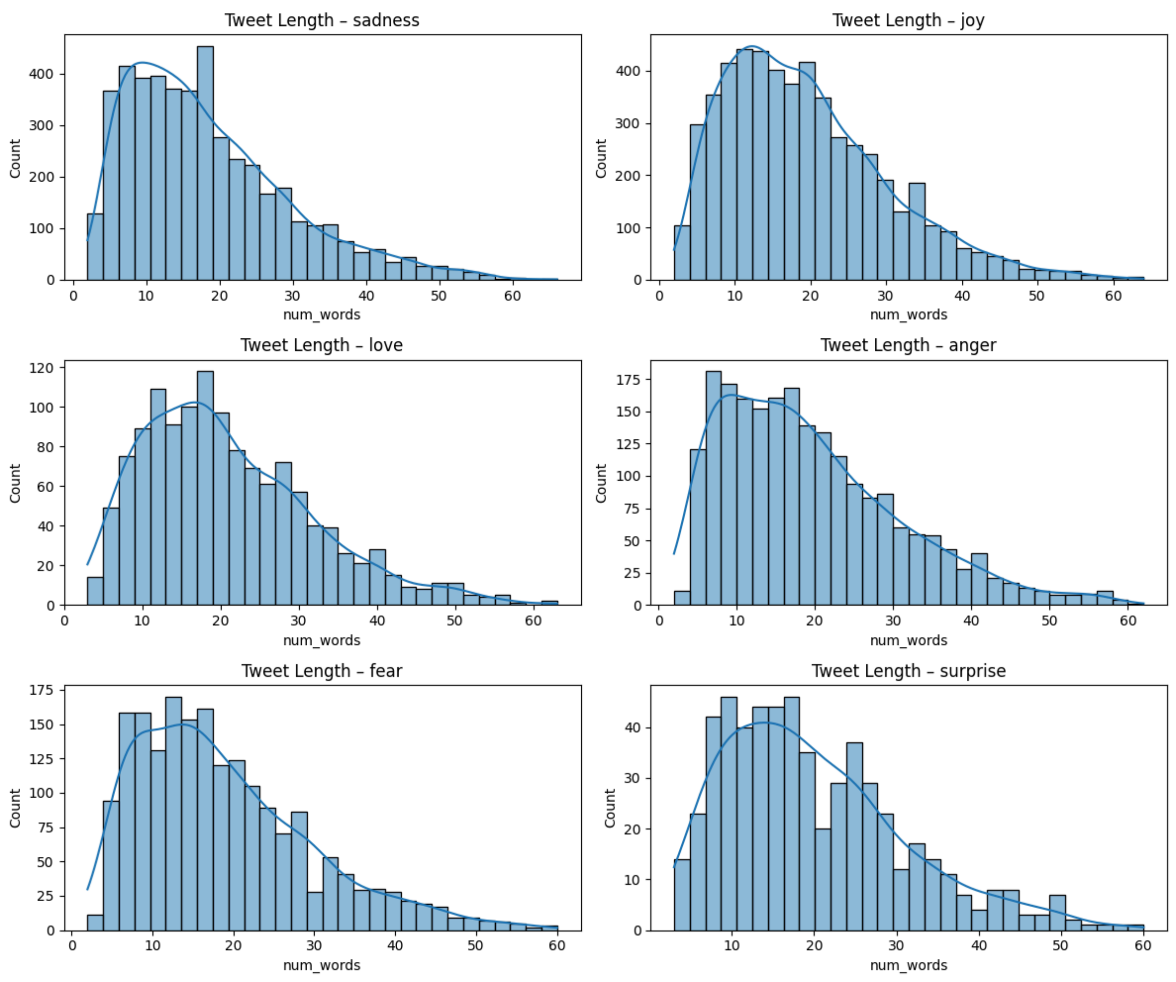
Overall, all emotions fall within a **tight range (18–21 words)**, confirming that tweets are consistently short and supporting a moderate max sequence length (e.g., 40–50 tokens) for modeling.

Tweet Length Distribution Visualization

In [35]:
```
# Overall Tweet Length Distribution
plt.figure(figsize=(7,4))
sns.histplot(train_df["num_words"], bins=40, kde=True)
plt.title("Overall Tweet Length Distribution")
plt.xlabel("num_words")
plt.ylabel("Count")
plt.tight_layout()
plt.show()
```



In [36]:
```
# Tweet Length Distribution per Emotion Label
plt.figure(figsize=(12, 10))

for i, (lbl, emotion_name) in enumerate(EMOTION_DICT.items(), 1):
    subset = train_df[train_df["label"] == lbl]["num_words"]

    plt.subplot(3, 2, i)
    sns.histplot(subset, bins=30, kde=True)
    plt.title(f"Tweet Length – {emotion_name}")
    plt.xlabel("num_words")
    plt.ylabel("Count")

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



Quick Look — Word Length Analysis

Overall Interpretation

The dataset is dominated by **short tweets**, mostly between **8–20 words**, with a right-skewed tail up to ~60 words.
This means:

- Most emotional content is expressed **briefly**.
- **Very long tweets are rare**, so models don't need large max sequence lengths.
- Setting sequence length around **40–60 tokens** captures almost all samples.
- Short text makes TF-IDF, LSTM, and BERT all efficient to train.

Overall, emotion classification on this dataset is lightweight and well-bounded in length.

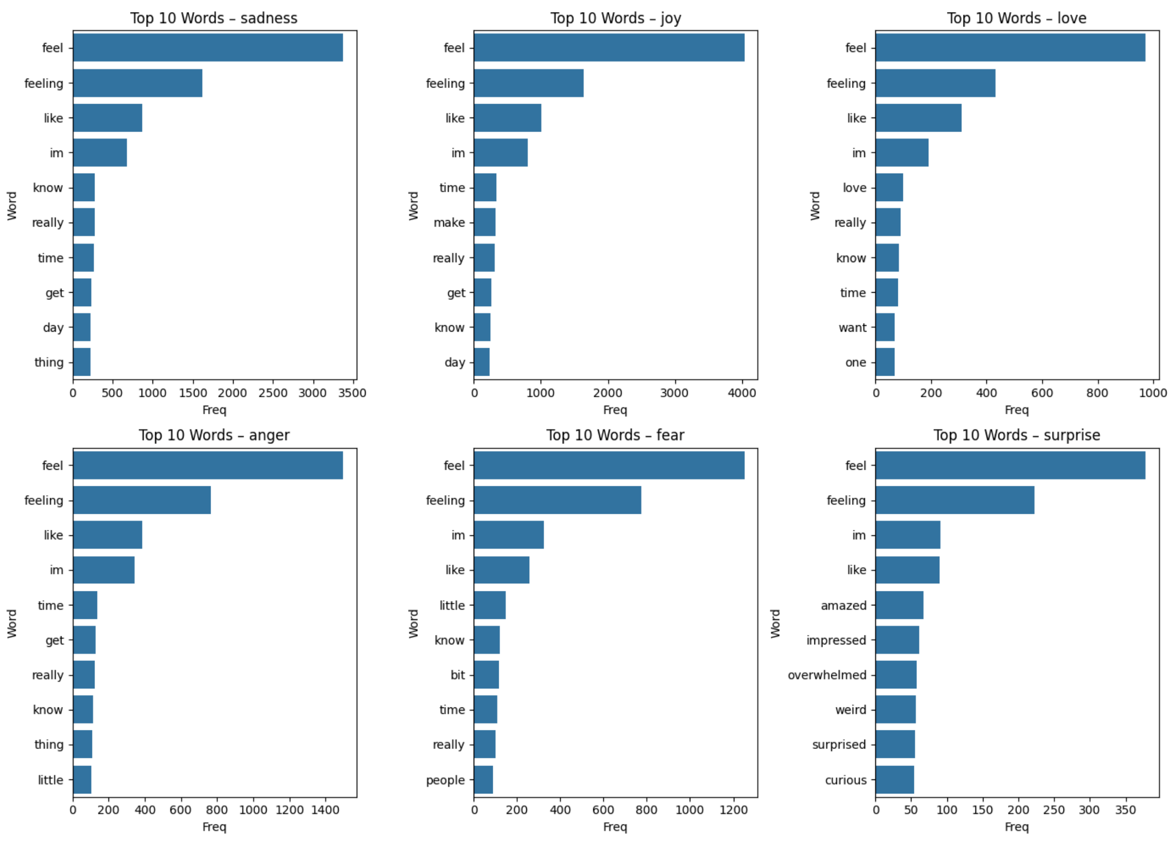
Quick Look — Per Emotion Label

- **Sadness:**: Slightly longer tweets; often **10–25 words** with reflective explanations.
- **Joy:**: Mostly short and spontaneous; concentrated around **8–18 words**.
- **Love:** More varied; many tweets between **12–25 words**, sometimes descriptive.
- **Anger:** Compact but with added context; common range **10–22 words**.
- **Fear:** Wider length spread; typically **10–25 words**, some longer descriptions.

- **Surprise:** Usually short reaction messages; often **8–16 words**.

Top Words per Emotion Class

In [37]:
```python
top_n = 10

plt.figure(figsize=(14, 10))

for i, (lbl, emotion_name) in enumerate(EMOTION_DICT.items(), 1):
    # Get cleaned tokens for this label
    subset = train_df[train_df["label"] == lbl]["clean_tokens"]

    # Flatten all token lists into one list
    tokens = []
    for tok_list in subset:
        tokens.extend(tok_list)

    # Top-N most common words
    counts = Counter(tokens).most_common(top_n)
    words  = [w for w, _ in counts]
    freqs  = [c for _, c in counts]

    # Plot
    plt.subplot(2, 3, i)
    sns.barplot(x=freqs, y=words, orient="h")
    plt.title(f"Top {top_n} Words – {emotion_name}")
    plt.xlabel("Freq")
    plt.ylabel("Word")

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



Quick Insights: Top 10 Words per Emotion

- **Sadness:** dominated by *feel, feeling, like, im, know* → reflective, low-energy expressions.
- **Joy:** similar structure but more positive verbs (*make, really*) → upbeat emotional tone.
- **Love:** includes more relational words (*love, want, one*) → affectionate, connection-focused.
- **Anger:** still uses *feel/feeling* but context is harsher → frustration, irritation.
- **Fear:** shows uncertainty (*little, bit, people*) → hesitation, vulnerability.
- **Surprise:** most distinct vocabulary (*amazed, impressed, overwhelmed*) → unexpected events, curiosity.

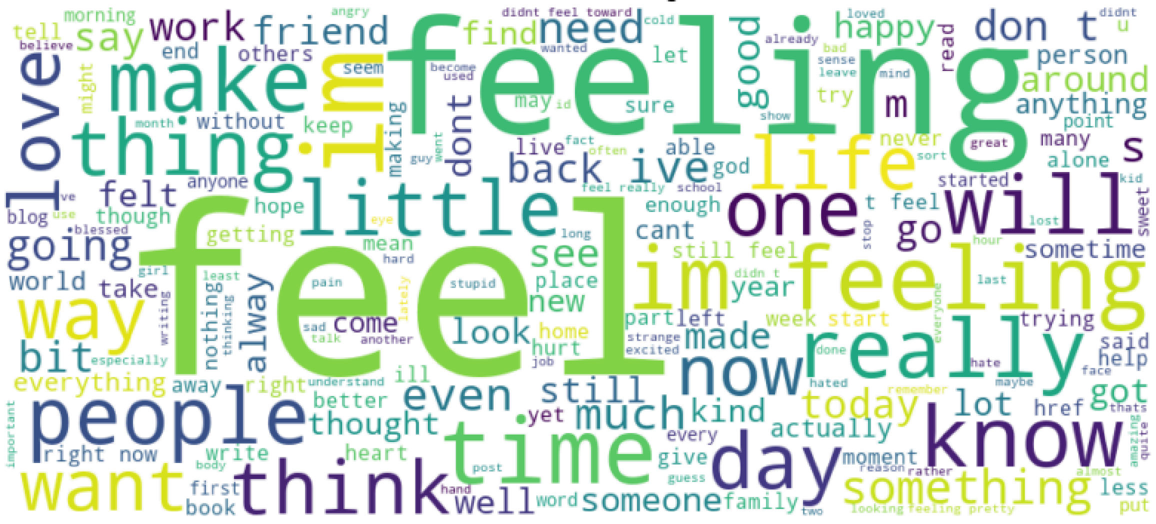
Overall: Many classes share core verbs like *feel/feeling*, but emotion-specific words help differentiate categories.

WordClouds

In [38]:
```python
def plot_wordcloud(texts, title):
    text = " ".join(texts)
    wc = WordCloud(width=900, height=400, background_color="white").gener
    plt.figure(figsize=(10,5))
    plt.imshow(wc, interpolation="bilinear")
    plt.axis("off")
    plt.title(title)
    plt.show()
```

In [39]:
```python
# Overall
plot_wordcloud(train_df["text"].tolist(), "WordCloud – All Training Tweet
```

WordCloud – All Training Tweets

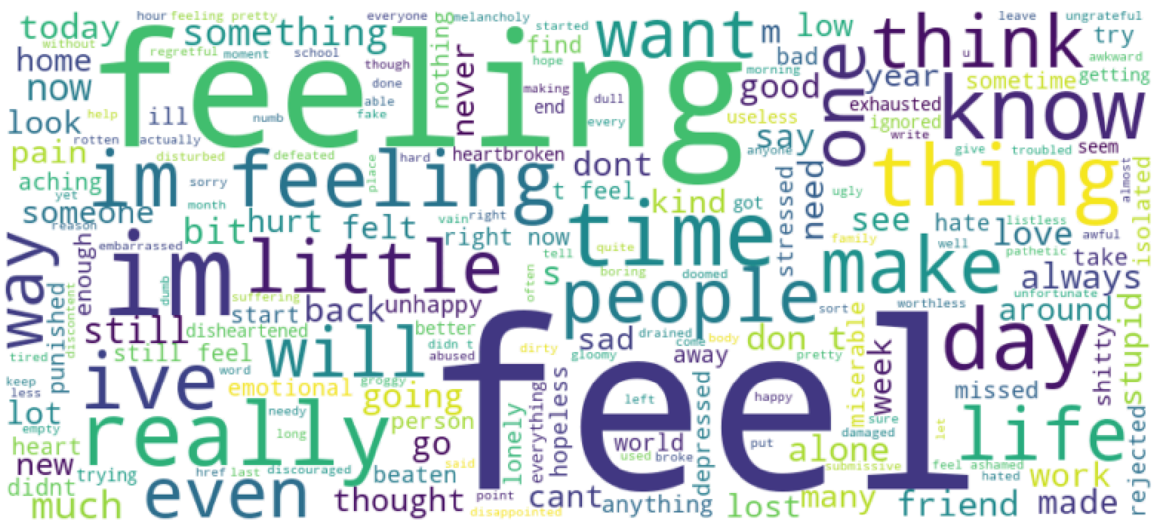


Interpretation

The overall word cloud highlights the most frequent terms across all tweets. Words like **feel**, **im**, **make**, **time**, **love**, and **know** dominate, reflecting the emotional and personal nature of Twitter content. These frequent terms provide an initial sense of common themes before emotion-specific analysis.

In [40]:
```
# sadness class
lbl = 0
lbl_name = EMOTION_DICT[lbl]

subset = train_df[train_df["label"] == lbl]
plot_wordcloud(subset["text"].tolist(), f"WordCloud – {lbl}")
```

WordCloud – 0

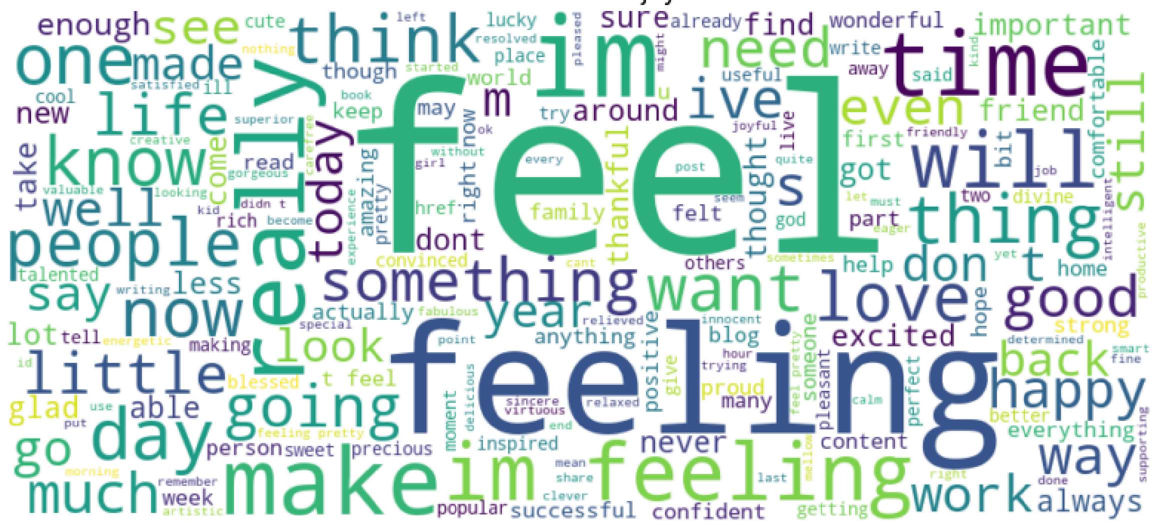


Interpretation (Sadness)

The sadness class is dominated by words expressing negative emotions and personal struggle. Terms like **feel**, **alone**, **hurt**, **pain**, **lost**, and **sad** appear frequently, reflecting themes of emotional distress, loneliness, and difficult life situations.

In [41]:
```
# joy class
lbl = 1
lbl_name = EMOTION_DICT[lbl]

subset = train_df[train_df["label"] == lbl]
plot_wordcloud(subset["text"].tolist(), f"WordCloud – {lbl_name}")
```

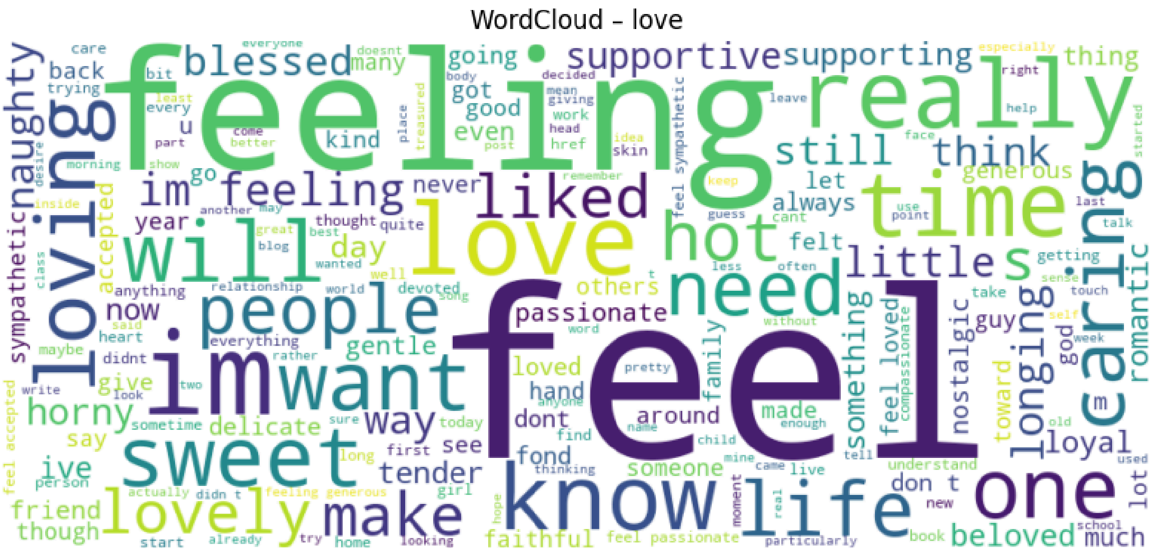
WordCloud – joy



Interpretation (Joy)

The joy class contains many positive and uplifting terms. Words such as **happy**, **good**, **love**, **excited**, **thankful**, and **glad** frequently appear, reflecting themes of celebration, appreciation, and positive life events.

In [42]:
```python
# love class
lbl = 2
lbl_name = EMOTION_DICT[lbl]

subset = train_df[train_df["label"] == lbl]
plot_wordcloud(subset["text"].tolist(), f"WordCloud – {lbl_name}")
```

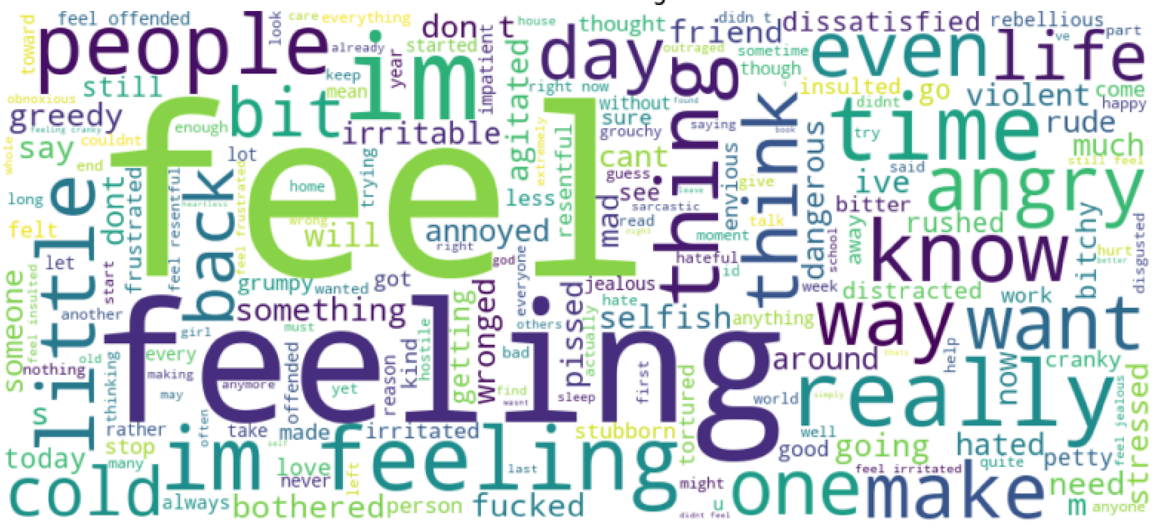


Interpretation (Love)

The love class features warm and affectionate words such as **love**, **lovely**, **caring**, **sweet**, **beloved**, and **romantic**. The vocabulary reflects themes of affection, connection, tenderness, and positive emotional bonding.

In [43]:
```python
# anger class
lbl = 3
lbl_name = EMOTION_DICT[lbl]

subset = train_df[train_df["label"] == lbl]
plot_wordcloud(subset["text"].tolist(), f"WordCloud – {lbl_name}")
```

WordCloud – anger

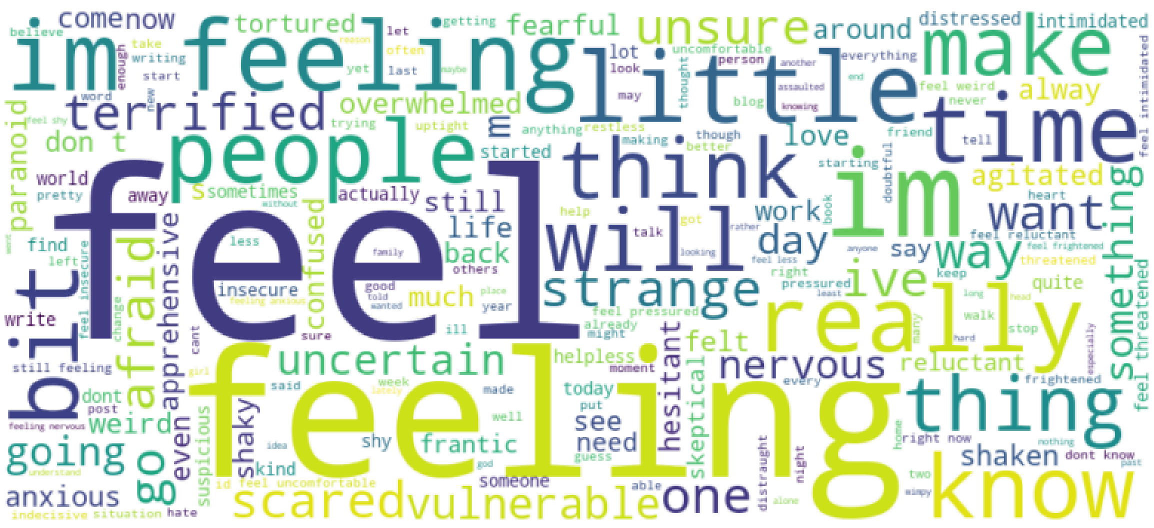


Interpretation (Anger)

The anger class contains many negative and intense terms such as **angry**, **annoyed**, **pissed**, **frustrated**, **irritated**, and **bitter**. The vocabulary reflects themes of conflict, frustration, and emotional tension.

In [44]:
```python
# fear class
lbl = 4
lbl_name = EMOTION_DICT[lbl]

subset = train_df[train_df["label"] == lbl]
plot_wordcloud(subset["text"].tolist(), f"WordCloud – {lbl_name}")
```

WordCloud – fear

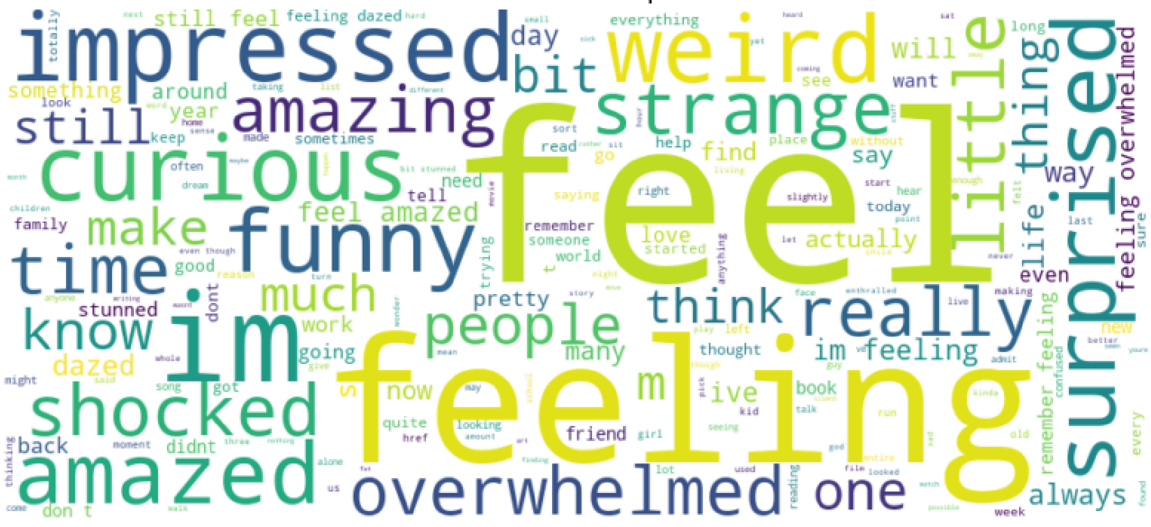


Interpretation (Fear)

The fear class is dominated by words such as **afraid**, **scared**, **terrified**, **unsure**, **nervous**, and **anxious**, indicating strong feelings of uncertainty, vulnerability, and emotional distress. Many terms reflect panic, insecurity, and a sense of being overwhelmed.

In [45]:
```
# surprise class
lbl = 5
lbl_name = EMOTION_DICT[lbl]

subset = train_df[train_df["label"] == lbl]
plot_wordcloud(subset["text"].tolist(), f"WordCloud – {lbl_name}")
```

WordCloud – surprise



Interpretation (Surprise)

The surprise class includes words such as **amazed**, **shocked**, **curious**, **weird**, **impressed**, and **overwhelmed**, reflecting reactions to unexpected events. The language shows a mix of astonishment, curiosity, and positive surprise.

3.3 Preparing Clean Text and Labels for Modeling

In [46]:
```
# BERT uses raw text
X_train_raw = train_df["text"].astype(str).tolist()
X_dev_raw   = dev_df["text"].astype(str).tolist()
X_test_raw  = test_df["text"].astype(str).tolist()
```

In [47]:
```
# Cleaned text lists
X_train_clean = train_df["clean_text"].tolist()
X_dev_clean   = dev_df["clean_text"].tolist()
X_test_clean  = test_df["clean_text"].tolist()
```

In [48]:
```
y_train = train_df["label_id"].values
y_dev   = dev_df["label_id"].values
y_test  = test_df["label_id"].values
```

3.4 Oversampling

In [49]:
```
print("Original label counts:", Counter(y_train))
```

```
Original label counts: Counter({np.int64(1): 5362, np.int64(0): 4666, np.i
nt64(3): 2159, np.int64(4): 1937, np.int64(2): 1304, np.int64(5): 572})
```

```
In [50]: ros = RandomOverSampler(random_state=SEED)
         idx_dummy = np.arange(len(X_train_clean)).reshape(-1,1)
         idx_res, y_res = ros.fit_resample(idx_dummy, y_train)
         X_train_clean_bal = [X_train_clean[i[0]] for i in idx_res]
         y_train_bal = y_res
```

```
In [51]: print("Oversampled label counts:", Counter(y_train_bal))
```

```
Oversampled label counts: Counter({np.int64(0): 5362, np.int64(3): 5362, n
p.int64(2): 5362, np.int64(5): 5362, np.int64(4): 5362, np.int64(1): 536
2})
```

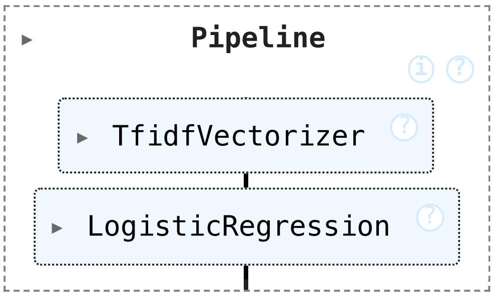
4. Model Training

4.1 TF-IDF + Logistic Regression

```
In [52]: tfidf_lr = Pipeline([
             ("tfidf", TfidfVectorizer(
                 max_features=20000,
                 ngram_range=(1,2),
             )),
             ("clf", LogisticRegression(
                 max_iter=1000,
                 class_weight="balanced",
                 n_jobs=-1,
             )),
         ])
```

```
In [53]: tfidf_lr.fit(X_train_clean_bal, y_train_bal)
```

Out[53]:



```
In [54]: y_pred_lr = tfidf_lr.predict(X_test_clean)
```

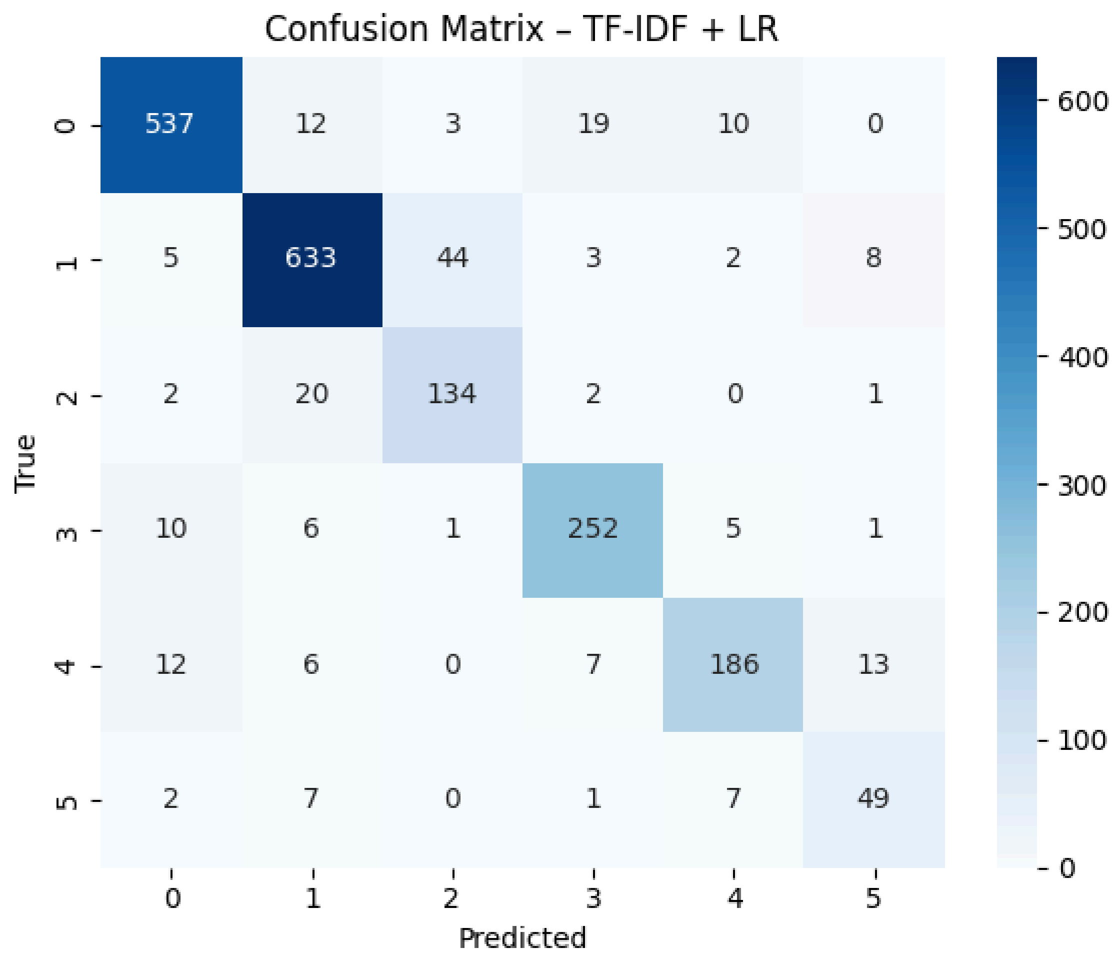
```
In [55]: print("Result in TF-IDF + Logistic Regression:")
         print("Accuracy:", accuracy_score(y_test, y_pred_lr))
         print("Macro F1:", f1_score(y_test, y_pred_lr, average="macro"))
         print("Weighted F1:", f1_score(y_test, y_pred_lr, average="weighted"))
```

```
Result in TF-IDF + Logistic Regression:
Accuracy: 0.8955
Macro F1: 0.8512673279207412
Weighted F1: 0.8964496130401985
```

```
In [56]: print("\nClassification Report:\n")
         print(
             classification_report(
                 y_test,
                 y_pred_lr,
                 target_names=EMOTION_DICT.values()
             )
         )
```

```
Classification Report:

              precision    recall  f1-score   support

     sadness       0.95      0.92      0.93       581
         joy       0.93      0.91      0.92       695
        love       0.74      0.84      0.79       159
       anger       0.89      0.92      0.90       275
        fear       0.89      0.83      0.86       224
    surprise       0.68      0.74      0.71        66

    accuracy                           0.90      2000
   macro avg       0.84      0.86      0.85      2000
weighted avg       0.90      0.90      0.90      2000
```

In [57]:
```
cm_lr = confusion_matrix(y_test, y_pred_lr)
plt.figure(figsize=(6,5))
sns.heatmap(cm_lr, annot=True, fmt="d", cmap="Blues",
            xticklabels=label_encoder.classes_,
            yticklabels=label_encoder.classes_)
plt.title("Confusion Matrix – TF-IDF + LR")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.show()
```



Performance Summary (TF-IDF + Logistic Regression)

- The TF-IDF + Logistic Regression model achieves **~90% overall accuracy**, with a **macro F1-score of 0.85** and **weighted F1-score of 0.90**, indicating a strong and well-balanced baseline.
- Core emotion classes (*sadness, joy, anger, fear*) are classified very effectively, all reaching **F1-scores between 0.86 and 0.93**, showing that linear models can capture clear lexical emotion cues.
- Performance drops for minority and more ambiguous classes:
  - *Love* (F1 = **0.79**) is frequently confused with *joy*, reflecting lexical overlap.
  - *Surprise* (F1 = **0.71**) remains the weakest class due to limited samples and less distinctive wording.
- The confusion matrix highlights predictable misclassifications between semantically close emotions (e.g., *joy–love*, *fear–surprise*), rather than random errors.

Overall, TF-IDF + Logistic Regression provides a **robust, interpretable, and competitive baseline**, setting a high reference point for comparing more complex neural and transformer-based models.

4.2 LSTM model

Word2Vec + tokenizer

In [58]:
```
train_tokens = train_df["clean_tokens"].tolist()
dev_tokens   = dev_df["clean_tokens"].tolist()
test_tokens  = test_df["clean_tokens"].tolist()

all_tokens = train_tokens + dev_tokens + test_tokens
```

In [59]:
```
w2v = Word2Vec(
    sentences=all_tokens,
    vector_size=100,
    window=5,
    min_count=2,
    workers=4,
    sg=1
)

print("Word2Vec vocab size:", len(w2v.wv))
```

```
Exception ignored in: 'gensim.models.word2vec_inner.our_dot_float'
Word2Vec vocab size: 7537
```

In [60]:
```
# Keras tokenizer
MAX_WORDS = 20000
MAX_SEQ_LEN = 40
```

In [61]:
```
tokenizer = Tokenizer(num_words=MAX_WORDS, oov_token="<UNK>")
tokenizer.fit_on_texts([" ".join(t) for t in all_tokens])
```

In [62]:
```
def to_seq(token_list):
    text = " ".join(token_list)
    seq = tokenizer.texts_to_sequences([text])[0]
    return seq
```

In [63]:
```
# Pad all sequences
def pad_all(tokens_list):
    seqs = tokenizer.texts_to_sequences([" ".join(t) for t in tokens_list
    return pad_sequences(seqs, maxlen=MAX_SEQ_LEN, padding="post", trunca
```

In [64]:
```
X_train_seq = pad_all(train_tokens)
X_dev_seq   = pad_all(dev_tokens)
X_test_seq  = pad_all(test_tokens)
```

In [65]:
```
EMB_DIM = 100
```

In [66]:
```
# Build embedding matrix
vocab_size = min(MAX_WORDS, len(tokenizer.word_index) + 1)
embedding_matrix = np.random.normal(0, 0.01, (vocab_size, EMB_DIM))

for w, i in tokenizer.word_index.items():
    if i < vocab_size and w in w2v.wv:
        embedding_matrix[i] = w2v.wv[w]
```

LSTM model Training

In [67]:
```
LSTM_EPOCH = 30
```

In [68]:
```
tf.keras.backend.clear_session()

lstm = tf.keras.Sequential([
    layers.Embedding(input_dim=vocab_size, output_dim=128, input_length=M
    layers.LSTM(128, dropout=0.2, recurrent_dropout=0.2),
    layers.Dropout(0.3),
    layers.Dense(64, activation="relu", kernel_regularizer=regularizers.l
    layers.Dropout(0.2),
    layers.Dense(NUM_CLASSES, activation="softmax"),
])

lstm.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

lstm.summary()
```

```
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/keras/sr
c/layers/core/embedding.py:97: UserWarning: Argument `input_length` is dep
recated. Just remove it.
  warnings.warn(
```

Model: "sequential"

Layer (type)	Output Shape	I
embedding (Embedding)	?	0 (ur
lstm (LSTM)	?	0 (ur
dropout (Dropout)	?	
dense (Dense)	?	0 (ur
dropout_1 (Dropout)	?	
dense_1 (Dense)	?	0 (ur

Total params: 0 (0.00 B)
Trainable params: 0 (0.00 B)
Non-trainable params: 0 (0.00 B)

In [69]:
```
# Compute class weights to handle label imbalance during training.
weights = compute_class_weight("balanced", classes=np.unique(y_train), y=
class_weights = {i: w for i, w in enumerate(weights)}
```

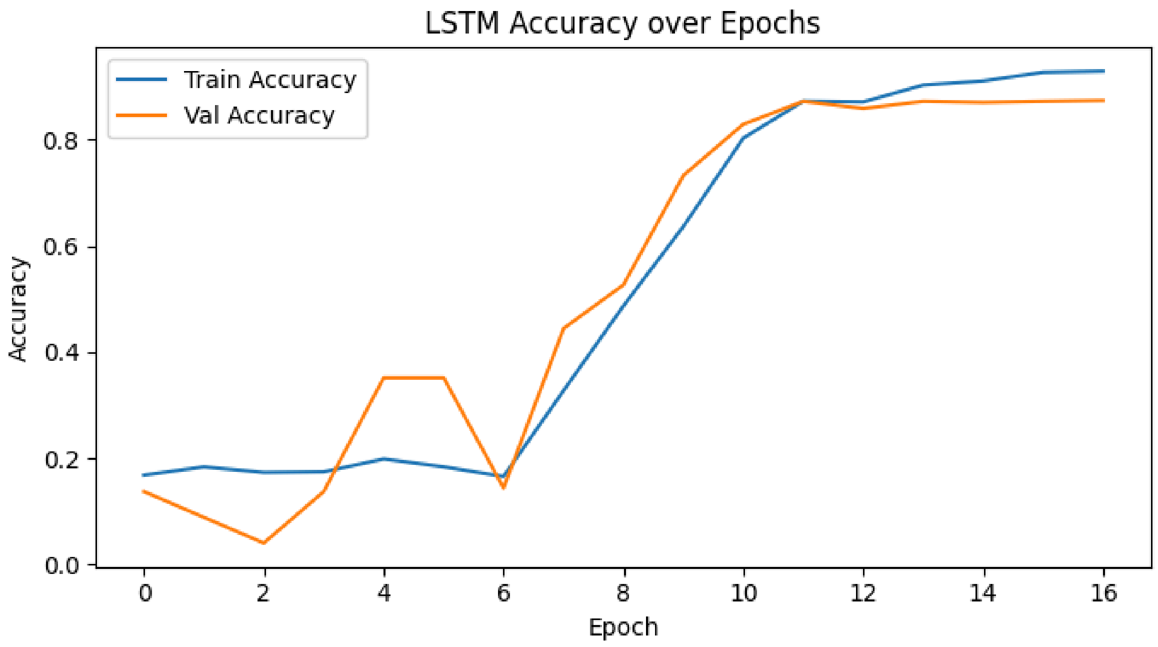
In [70]:
```
callbacks = [
    ReduceLROnPlateau(
        monitor="val_loss",
        factor=0.5,
        patience=3,
        min_lr=1e-5,
        verbose=1
    ),
    EarlyStopping(
        monitor="val_loss",
        patience=5,
        min_delta=1e-3,
        restore_best_weights=True,
        verbose=1
    ),
]
```

In [71]:
```
history = lstm.fit(
    X_train_seq, y_train,
    validation_data=(X_dev_seq, y_dev),
    batch_size=64,
    epochs=LSTM_EPOCH,
    class_weight=class_weights,
    callbacks=callbacks,
    verbose=1
)

y_pred_lstm = np.argmax(lstm.predict(X_test_seq), axis=1)

print("Results in Word2Vec + LSTM:")
print("Accuracy :", accuracy_score(y_test, y_pred_lstm))
print("Macro F1 :", f1_score(y_test, y_pred_lstm, average="macro"))
print("Weighted :", f1_score(y_test, y_pred_lstm, average="weighted"))
```

```
Epoch 1/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 23s 81ms/step - accuracy: 0.1686 - loss: 1.80
14 - val_accuracy: 0.1375 - val_loss: 1.7978 - learning_rate: 0.0010
Epoch 2/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 21s 85ms/step - accuracy: 0.1841 - loss: 1.79
88 - val_accuracy: 0.0890 - val_loss: 1.7977 - learning_rate: 0.0010
Epoch 3/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 19s 76ms/step - accuracy: 0.1736 - loss: 1.79
73 - val_accuracy: 0.0405 - val_loss: 1.7961 - learning_rate: 0.0010
Epoch 4/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 19s 78ms/step - accuracy: 0.1748 - loss: 1.79
60 - val_accuracy: 0.1375 - val_loss: 1.8001 - learning_rate: 0.0010
Epoch 5/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 20s 80ms/step - accuracy: 0.1989 - loss: 1.79
50 - val_accuracy: 0.3520 - val_loss: 1.7907 - learning_rate: 0.0010
Epoch 6/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 20s 81ms/step - accuracy: 0.1842 - loss: 1.79
41 - val_accuracy: 0.3520 - val_loss: 1.7900 - learning_rate: 0.0010
Epoch 7/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 20s 81ms/step - accuracy: 0.1659 - loss: 1.78
98 - val_accuracy: 0.1440 - val_loss: 1.7331 - learning_rate: 0.0010
Epoch 8/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 20s 78ms/step - accuracy: 0.3276 - loss: 1.45
12 - val_accuracy: 0.4445 - val_loss: 1.1855 - learning_rate: 0.0010
Epoch 9/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 19s 76ms/step - accuracy: 0.4873 - loss: 1.07
30 - val_accuracy: 0.5270 - val_loss: 0.9487 - learning_rate: 0.0010
Epoch 10/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 19s 76ms/step - accuracy: 0.6369 - loss: 0.76
24 - val_accuracy: 0.7330 - val_loss: 0.7335 - learning_rate: 0.0010
Epoch 11/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 18s 73ms/step - accuracy: 0.8033 - loss: 0.57
40 - val_accuracy: 0.8295 - val_loss: 0.5870 - learning_rate: 0.0010
Epoch 12/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 19s 74ms/step - accuracy: 0.8727 - loss: 0.41
74 - val_accuracy: 0.8725 - val_loss: 0.4967 - learning_rate: 0.0010
Epoch 13/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 19s 74ms/step - accuracy: 0.8714 - loss: 0.44
07 - val_accuracy: 0.8590 - val_loss: 0.5400 - learning_rate: 0.0010
Epoch 14/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 18s 73ms/step - accuracy: 0.9032 - loss: 0.35
70 - val_accuracy: 0.8725 - val_loss: 0.5012 - learning_rate: 0.0010
Epoch 15/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 0s 70ms/step - accuracy: 0.9083 - loss: 0.321
4
Epoch 15: ReduceLROnPlateau reducing learning rate to 0.000500000023748725
7.
250/250 ━━━━━━━━━━━━━━━━━━━━ 18s 72ms/step - accuracy: 0.9110 - loss: 0.31
73 - val_accuracy: 0.8705 - val_loss: 0.5083 - learning_rate: 0.0010
Epoch 16/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 15s 60ms/step - accuracy: 0.9272 - loss: 0.27
17 - val_accuracy: 0.8725 - val_loss: 0.5048 - learning_rate: 5.0000e-04
Epoch 17/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 15s 59ms/step - accuracy: 0.9293 - loss: 0.25
65 - val_accuracy: 0.8740 - val_loss: 0.4998 - learning_rate: 5.0000e-04
Epoch 17: early stopping
Restoring model weights from the end of the best epoch: 12.
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step
Results in Word2Vec + LSTM:
Accuracy : 0.8645
```

```
Macro F1 : 0.8107109632724345
Weighted : 0.872653676207444
```

In [72]:
```
plt.figure(figsize=(7,4))
plt.plot(history.history["accuracy"], label="Train Accuracy")
plt.plot(history.history["val_accuracy"], label="Val Accuracy")
plt.title("LSTM Accuracy over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.tight_layout()
plt.show()
```



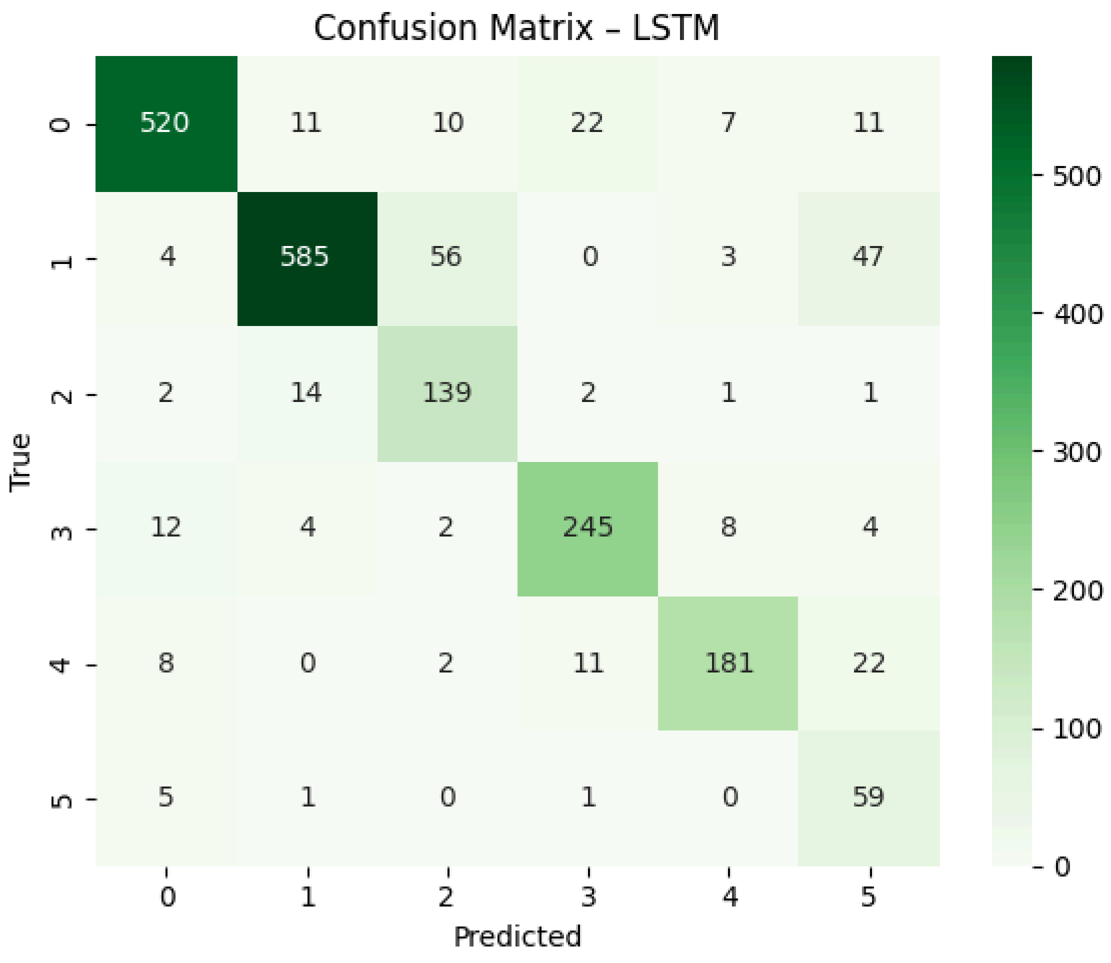
Quick Look — LSTM Accuracy over Epochs

- Training and validation accuracy start very low, then increase sharply after a few epochs, indicating delayed but effective learning.
- Validation accuracy closely follows training accuracy, suggesting **no severe overfitting**.
- Performance stabilizes around **~0.85–0.88 validation accuracy**, showing reasonable generalization.
- Learning dynamics are less stable early on, reflecting sensitivity to initialization and data imbalance.

In [73]:
```
print("\nClassification Report:\n")
print(
    classification_report(
    y_test,
    y_pred_lstm,
    target_names=EMOTION_DICT.values()
))
```

```
Classification Report:

              precision    recall  f1-score   support

     sadness       0.94      0.90      0.92       581
         joy       0.95      0.84      0.89       695
        love       0.67      0.87      0.76       159
       anger       0.87      0.89      0.88       275
        fear       0.91      0.81      0.85       224
    surprise       0.41      0.89      0.56        66

    accuracy                           0.86      2000
   macro avg       0.79      0.87      0.81      2000
weighted avg       0.89      0.86      0.87      2000
```

In [74]:
```python
cm_lstm = confusion_matrix(y_test, y_pred_lstm)
plt.figure(figsize=(6,5))
sns.heatmap(
    cm_lstm, annot=True, fmt="d", cmap="Greens",
    xticklabels=label_encoder.classes_,
    yticklabels=label_encoder.classes_,
)
plt.title("Confusion Matrix – LSTM")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.show()
```



Performance Summary (LSTM)

- The LSTM model achieves **86% overall accuracy**, with a **macro F1-score of 0.81** and **weighted F1-score of 0.87**, indicating solid but uneven performance across classes.
- Major emotions (*sadness, joy, anger, fear*) are recognized reliably, with F1-scores ranging from **0.85–0.92**, showing effective sequence modeling of common patterns.
- Minority classes remain challenging:
  - *Love* shows moderate performance (F1 = **0.76**), with confusion mainly toward *joy*.
  - *Surprise* has the weakest performance (F1 = **0.56**), driven by very low precision despite high recall, indicating frequent false positives.
- The confusion matrix reveals overlaps between semantically close emotions (e.g., *joy–love*, *fear–surprise*), suggesting limited discrimination for subtle emotional cues.

Overall, the LSTM provides **reasonable generalization** and improves over simpler sequence baselines, but remains constrained by class imbalance and limited contextual depth, especially for rare emotions.

4.3 BERT dataset

BERT dataset

In [75]:
```
BERT_MODEL = "bert-base-uncased"
bert_tokenizer = AutoTokenizer.from_pretrained(BERT_MODEL)
```

In [76]:
```
class EmotionDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len=64):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __getitem__(self, idx):
        encoding = self.tokenizer(
            self.texts[idx],
            max_length=self.max_len,
            padding="max_length",
            truncation=True,
            return_tensors="pt"
        )
        item = {k: v.squeeze(0) for k, v in encoding.items()}
        item["labels"] = torch.tensor(int(self.labels[idx]))
        return item

    def __len__(self):
        return len(self.texts)

train_dataset = EmotionDataset(X_train_raw, y_train, bert_tokenizer)
dev_dataset   = EmotionDataset(X_dev_raw, y_dev, bert_tokenizer)
test_dataset  = EmotionDataset(X_test_raw, y_test, bert_tokenizer)
```

Fine-tuning BERT

In [77]:
```
BERT_EPOCH=5
```

In [78]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device:", device)

bert_model = AutoModelForSequenceClassification.from_pretrained(
    BERT_MODEL, num_labels=NUM_CLASSES
).to(device)
```

```
Using device: cpu
```

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

In [79]:
```
def compute_metrics(pred):
    logits, labels = pred
    preds = np.argmax(logits, axis=1)
    return {
        "accuracy": accuracy_score(labels, preds),
        "f1_macro": f1_score(labels, preds, average="macro"),
        "f1_weighted": f1_score(labels, preds, average="weighted"),
    }
```

In [80]:
```
args = TrainingArguments(
    output_dir="./bert_out",
    num_train_epochs=BERT_EPOCH,
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=32,
    weight_decay=0.01,
    logging_steps=50,
)
```

In [81]:
```
trainer = Trainer(
    model=bert_model,
    args=args,
    train_dataset=train_dataset,
    eval_dataset=dev_dataset,
    compute_metrics=compute_metrics,
)

trainer.train()
```

```
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/torch/ut
ils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as t
rue but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
```

[5000/5000 1:01:24, Epoch 5/5]

Step	Training Loss
50	1.653600
100	1.371900
150	1.223600
200	0.984700
250	0.720700
300	0.518900
350	0.457700
400	0.387900
450	0.342800
500	0.326000
550	0.291400
600	0.244900
650	0.317300
700	0.215500
750	0.243800
800	0.255600
850	0.254500
900	0.188700
950	0.240200
1000	0.250200
1050	0.136800
1100	0.131600
1150	0.194500
1200	0.161200
1250	0.153500
1300	0.165100
1350	0.136800
1400	0.149400
1450	0.140500
1500	0.201000
1550	0.124000
1600	0.140500
1650	0.161200

Step	Training Loss
1700	0.145900
1750	0.154700
1800	0.135700
1850	0.172700
1900	0.146900
1950	0.111400
2000	0.143400
2050	0.125300
2100	0.104400
2150	0.117500
2200	0.099900
2250	0.094700
2300	0.116600
2350	0.121500
2400	0.123800
2450	0.124000
2500	0.099700
2550	0.101600
2600	0.080700
2650	0.097400
2700	0.115000
2750	0.083500
2800	0.074100
2850	0.137200
2900	0.113800
2950	0.096900
3000	0.093700
3050	0.063000
3100	0.085000
3150	0.070200
3200	0.079000
3250	0.044400
3300	0.057900
3350	0.087300

Step	Training Loss
3400	0.128600
3450	0.058700
3500	0.077200
3550	0.081400
3600	0.066700
3650	0.095800
3700	0.113700
3750	0.055400
3800	0.091100
3850	0.070500
3900	0.067200
3950	0.075300
4000	0.064700
4050	0.032200
4100	0.049400
4150	0.042500
4200	0.056500
4250	0.050700
4300	0.040900
4350	0.070400
4400	0.042800
4450	0.054900
4500	0.048900
4550	0.070900
4600	0.049800
4650	0.072700
4700	0.046500
4750	0.041500
4800	0.045000
4850	0.078200
4900	0.056200
4950	0.040800
5000	0.039300

```
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/torch/ut
ils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as t
rue but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/torch/ut
ils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as t
rue but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/torch/ut
ils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as t
rue but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/torch/ut
ils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as t
rue but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
```

Out[81]:
```
TrainOutput(global_step=5000, training_loss=0.1818149274110794, metrics=
{'train_runtime': 3685.1428, 'train_samples_per_second': 21.709, 'train_
steps_per_second': 1.357, 'total_flos': 2631205048320000.0, 'train_los
s': 0.1818149274110794, 'epoch': 5.0})
```

In [82]:
```
outputs = trainer.predict(test_dataset)
y_pred_bert = np.argmax(outputs.predictions, axis=1)
```

```
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/torch/ut
ils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as t
rue but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
```

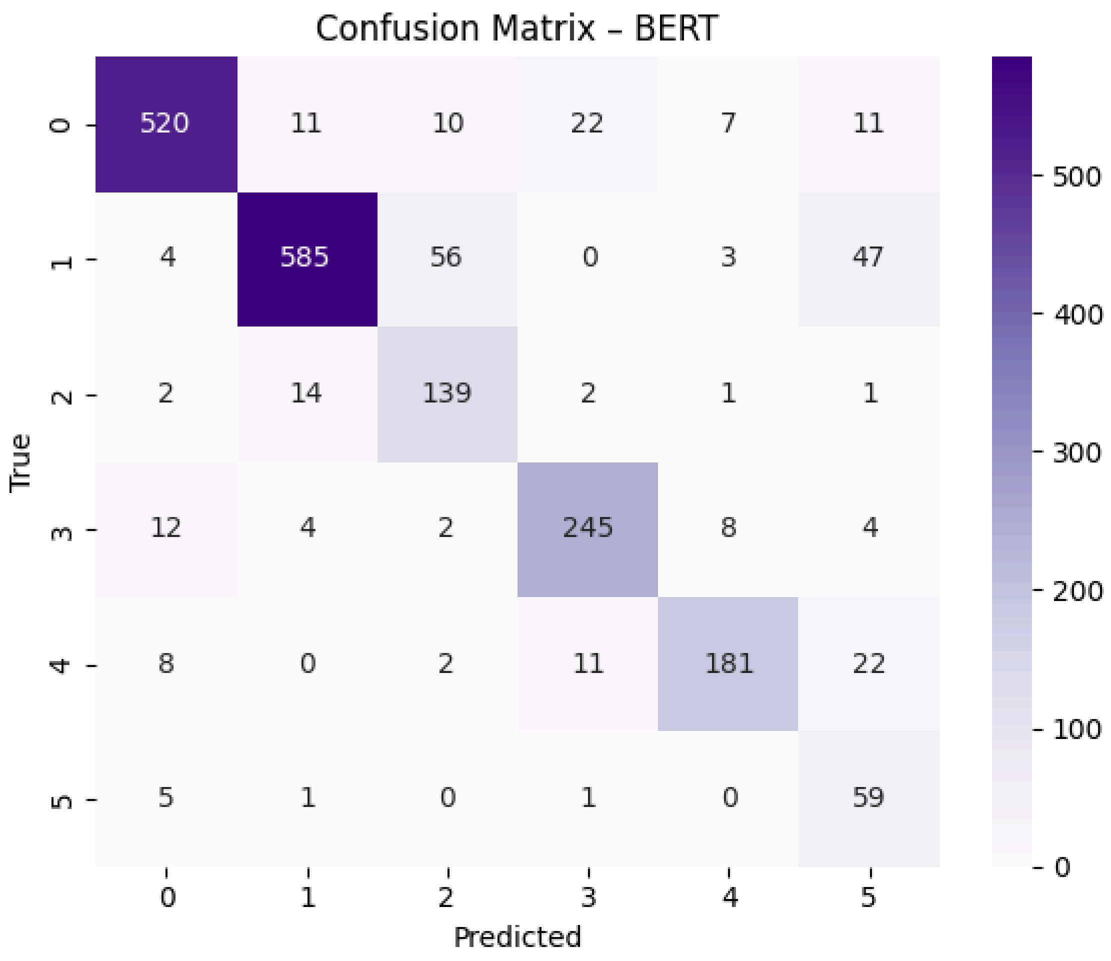
In [83]:
```
outputs = trainer.predict(test_dataset)
y_pred_bert = np.argmax(outputs.predictions, axis=1)

print("BERT Fine-Tuning")
print("Accuracy :", accuracy_score(y_test, y_pred_bert))
print("Macro F1 :", f1_score(y_test, y_pred_bert, average="macro"))
print("Weighted :", f1_score(y_test, y_pred_bert, average="weighted"))
```

```
/Users/taanhtuan/miniconda3/envs/nlp/lib/python3.10/site-packages/torch/ut
ils/data/dataloader.py:692: UserWarning: 'pin_memory' argument is set as t
rue but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
BERT Fine-Tuning
Accuracy : 0.9255
Macro F1 : 0.8799415441877375
Weighted : 0.9255455141662476
```

In [84]:
```
print(
    classification_report(
        y_test,
        y_pred_bert,
        target_names=EMOTION_DICT.values()
    )
)
```

```
              precision    recall  f1-score   support

     sadness       0.96      0.97      0.96       581
         joy       0.95      0.94      0.94       695
        love       0.79      0.82      0.81       159
       anger       0.95      0.91      0.93       275
        fear       0.88      0.91      0.90       224
    surprise       0.76      0.71      0.73        66

    accuracy                           0.93      2000
   macro avg       0.88      0.88      0.88      2000
weighted avg       0.93      0.93      0.93      2000
```

In [92]:
```
cm_bert = confusion_matrix(y_test, y_pred_bert)
plt.figure(figsize=(6,5))
sns.heatmap(
    cm_lstm, annot=True, fmt="d", cmap="Purples",
    xticklabels=label_encoder.classes_,
    yticklabels=label_encoder.classes_,
)
plt.title("Confusion Matrix – BERT")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.show()
```



True \ Predicted	0	1	2	3	4	5
0	520	11	10	22	7	11
1	4	585	56	0	3	47
2	2	14	139	2	1	1
3	12	4	2	245	8	4
4	8	0	2	11	181	22
5	5	1	0	1	0	59

Performance Summary (BERT Fine-Tuning)

- The fine-tuned BERT model achieves **93% overall accuracy**, delivering the strongest performance among all tested models.
- A **macro F1-score of approximately 0.88** indicates well-balanced performance across emotion classes, including minority categories.
- Major emotions (*sadness, joy, anger, fear*) show **high precision and recall (F1 ≥ 0.90)**, reflecting BERT's strong ability to capture contextual meaning in short tweets.
- Minority classes (*love* and *surprise*) remain more challenging but still show **clear improvements** compared to TF-IDF + Logistic Regression and LSTM-based models.
- The confusion matrix reveals fewer misclassifications overall, with most remaining errors occurring between **semantically related emotions** (e.g., *joy* ↔ *love*, *fear* ↔ *surprise*).

Overall, BERT provides the **most robust and reliable** emotion classification performance, benefiting from pre-trained contextual representations that generalize well to noisy, informal social-media text.

5. Result and Conclusion

5.1 Model Evaluation

In [85]:
```
results = []
```

In [86]:
```
results.append({
        "Model": "TF-IDF + LR",
        "Accuracy": accuracy_score(y_test, y_pred_lr),
        "Macro F1": f1_score(y_test, y_pred_lr, average="macro"),
        "Weighted F1": f1_score(y_test, y_pred_lr, average="weighted"),
    })
```

In [87]:
```
results.append({
        "Model": "Word2Vec + LSTM",
        "Accuracy": accuracy_score(y_test, y_pred_lstm),
        "Macro F1": f1_score(y_test, y_pred_lstm, average="macro"),
        "Weighted F1": f1_score(y_test, y_pred_lstm, average="weighted"),
    })
```

In [88]:
```
results.append({
        "Model": "BERT Fine-Tuning",
        "Accuracy": accuracy_score(y_test, y_pred_bert),
        "Macro F1": f1_score(y_test, y_pred_bert, average="macro"),
        "Weighted F1": f1_score(y_test, y_pred_bert, average="weighted"),
    })
```

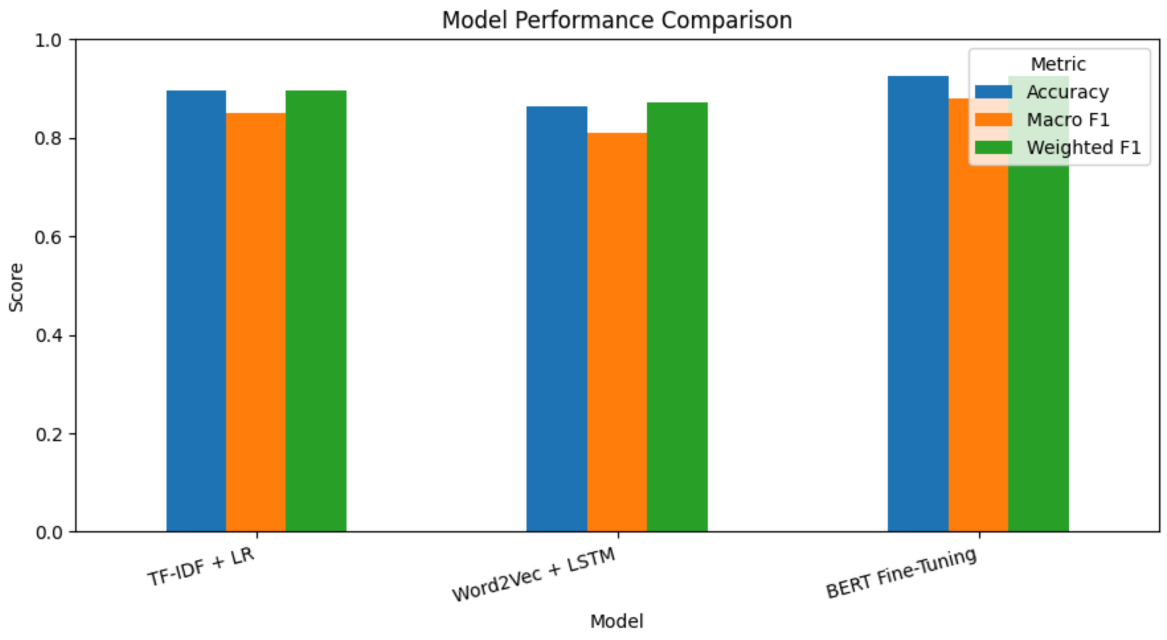
In [94]:
```
results_df = pd.DataFrame(results)
results_df
```

Out[94]:

	Model	Accuracy	Macro F1	Weighted F1
0	TF-IDF + LR	0.8955	0.851267	0.896450
1	Word2Vec + LSTM	0.8645	0.810711	0.872654
2	BERT Fine-Tuning	0.9255	0.879942	0.925546

In [95]:

```
metrics = ["Accuracy", "Macro F1", "Weighted F1"]
ax = results_df.set_index("Model")[metrics].plot(kind="bar", figsize=(9,

plt.title("Model Performance Comparison")
plt.xlabel("Model")
plt.ylabel("Score")
plt.ylim(0, 1)
plt.xticks(rotation=15, ha="right")
plt.legend(title="Metric")
plt.tight_layout()
plt.show()
```



5.1 Conclusion & Recommendation

Conclusion

This project presents a comprehensive comparison of **three NLP approaches for tweet-based emotion classification**: **TF-IDF + Logistic Regression**, **Word2Vec + LSTM**, and **BERT Fine-Tuning**.
All models were evaluated using **accuracy**, **macro F1**, and **weighted F1**, enabling assessment of both overall performance and robustness under class imbalance.

Model Comparison and Key Findings

- **TF-IDF + Logistic Regression**

- Achieved a strong baseline with **89.6% accuracy** and **macro F1 ≈ 0.85**.
- Demonstrated stable and consistent performance on high-frequency emotions such as *sadness*, *joy*, *anger*, and *fear*.
- Performance on minority classes (*love* and *surprise*) remained comparatively weaker, reflecting the limitations of bag-of-words representations in capturing subtle emotional semantics.
- Results confirm that classical machine-learning models, when paired with effective text preprocessing, remain highly competitive for short-text emotion classification.

- **Word2Vec + LSTM**
  - Achieved **86.5% accuracy** and **macro F1 ≈ 0.81**, improving semantic representation over TF-IDF but not surpassing the linear baseline.
  - Training exhibited delayed convergence, with low accuracy in early epochs followed by rapid improvement.
  - Results indicate sensitivity to class imbalance, dataset size, and initialization.
  - Highlights a key limitation of sequence-based neural models: without large-scale data or richer contextual signals, performance gains over simpler models may be limited.
- **BERT Fine-Tuning**
  - Delivered the strongest overall results:
    - **Accuracy: 92.6%**
    - **Macro F1: 0.88**
    - **Weighted F1: 0.93**
  - Achieved high and balanced precision–recall across both majority and minority emotion classes.
  - Confusion matrix analysis shows reduced cross-class confusion compared to TF-IDF and LSTM.
  - Effectively leveraged pre-trained contextual representations to capture nuanced emotional cues, idiomatic expressions, and semantic overlap common in tweets.

Overall Observations

- There is a clear performance progression from **bag-of-words → sequential embeddings → contextual transformers**.
- Models incorporating **pre-trained contextual knowledge** generalize better to short, noisy, and informal social-media text.
- While class imbalance affects all models, **BERT demonstrates the highest robustness** under these conditions.
- Increased model complexity alone (e.g., LSTM) does not guarantee better performance without sufficient data and context.

Final Conclusion

Overall, the experimental results confirm that **transformer-based architectures are best suited for emotion classification in social media contexts**.
While **TF-IDF + Logistic Regression** provides a strong and efficient baseline, and **Word2Vec + LSTM** offers moderate representational improvements, **BERT Fine-Tuning consistently delivers the most reliable and scalable performance**, making it the most suitable choice for real-world deployment.

Recommendation

- Use **BERT Fine-Tuning** as the primary production model.
- To further improve performance and robustness:
  - Experiment with **domain-specific transformers** (e.g., BERTweet).
  - Apply **class-aware loss functions** such as focal loss or class-balanced loss.
  - Incorporate additional signals (emojis, hashtags, metadata).
  - Optimize inference via **model distillation or quantization**.
  - Periodically retrain to adapt to evolving language patterns on social media.

8. References

- **Dataset Link**: https://www.kaggle.com/datasets/parulpandey/emotion-dataset/data
- **Github**: https://github.com/tuanTaAnh/Emotion-Classification

```
In [ ]:
```