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 host millions of short, informal messages expressing emotions, opinions, and reactions to real-world events. Businesses use these signals to improve customer understanding, track public sentiment, monitor brand reputation, and enhance user engagement.

However, manually analyzing emotional tone across massive text streams is infeasible and error-prone. Tweets are often noisy, contain slang, emojis, sarcasm, abbreviations, and exhibit highly imbalanced emotion distributions.

Developing an automated NLP system for emotion classification enables organizations to efficiently extract emotional insights and support:

- Customer service analytics
- Marketing and brand monitoring
- Mental-health trend detection
- Real-time public opinion analysis

Objective

Design and implement a supervised NLP pipeline that classifies the **emotional category** of a tweet.
The system should:

- Take a raw tweet as input.
- Output one of six emotion labels: **sadness, joy, love, anger, fear, surprise**.
- Apply appropriate preprocessing for noisy social-media text.
- Train and compare **three NLP models** studied in the course:
  - TF-IDF + Logistic Regression
  - Word2Vec Embeddings + BiLSTM
  - Transformer Model (BERT Fine-Tuning)
- Address class imbalance using resampling or class weights.
- Evaluate all models using class-balanced metrics and provide insights into strengths and limitations.

Dataset Description

Dataset

- **Source:** Public Twitter Emotion Dataset (Kaggle)
- **Total samples:** ≈ 20,000 tweets
- **Provided splits:**
  - `training.csv`
  - `validation.csv`
  - `test.csv`
- **Input:** Raw tweet text (may contain emojis, hashtags, slang)
- **Target:** One of six emotion categories

Class Distribution (Before Preprocessing)

The dataset is moderately imbalanced:

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

Minority classes (especially **love** and **surprise**) justify the use of balancing techniques and macro-averaged evaluation metrics.

High-Level Pipeline & Approach

- Perform **exploratory data analysis (EDA)**:
  - Class distribution
  - Tweet length distribution
  - Word clouds

    - Noise characteristics (emojis, hashtags, casing)
- Apply **NLP preprocessing** tailored for tweets:
    - Lowercasing
    - Tokenization
    - Preserving emojis and hashtags
    - Removing unnecessary symbols
    - Lemmatization
    - Optional handling of repeated characters and elongated words
- Build **three distinct feature-engineering + modeling pipelines**:
    1. **TF-IDF Vectorization + Logistic Regression**
    2. **Word2Vec Embeddings + BiLSTM Neural Network**
    3. **Transformer-based model (BERT)** with fine-tuning
- Address **class imbalance** using:
    - Random oversampling
    - Class-weighted training loss
    - Balanced mini-batches (for neural models)
- Evaluate all models on a held-out test set and compare performance using multiple metrics.

Evaluation Strategy

Because this is an imbalanced multi-class classification task, evaluation includes both standard and class-balanced metrics.

Primary Metrics

- **Macro-averaged F1-score**
  Treats all emotion classes equally and emphasizes minority-class performance.
- **Per-class Precision, Recall, and F1-score**
  Highlights difficult classes such as *love* and *surprise*.

Secondary Metrics

- **Overall Accuracy**
- **Weighted F1-score**
- **Confusion Matrix** for detailed error analysis
- **Validation loss curves** (for neural networks)

Expected Outcomes

- A complete end-to-end NLP system capable of classifying emotions in tweets.

- A comparison of classical machine-learning, word-embedding-based neural networks, and transformer-based models.
- Insights into the impact of preprocessing, feature engineering, and class imbalance on emotion detection performance.
- Recommendations for improving real-world emotion classification systems.

1. Imports and Config

In [1]:
```
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
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

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]:
```
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]:
```
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]:
```
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}%)")
```

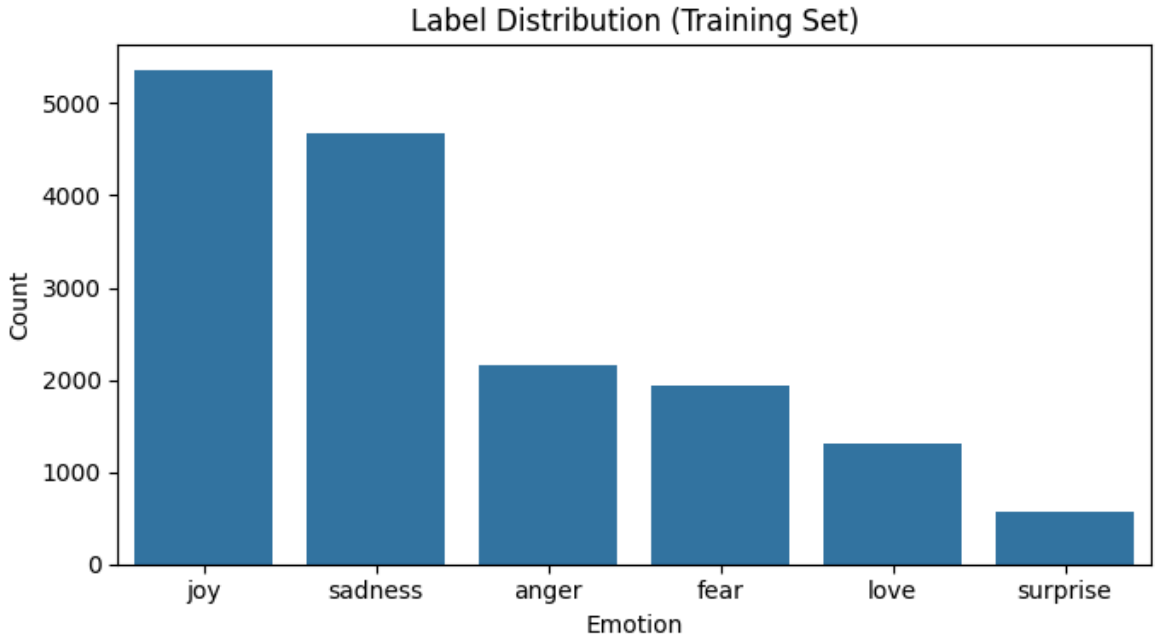
```
Label 1: 5,362 samples (33.51%)
Label 0: 4,666 samples (29.16%)
Label 3: 2,159 samples (13.49%)
Label 4: 1,937 samples (12.11%)
Label 2: 1,304 samples (8.15%)
Label 5: 572 samples (3.57%)
```

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_22578/98844256
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.

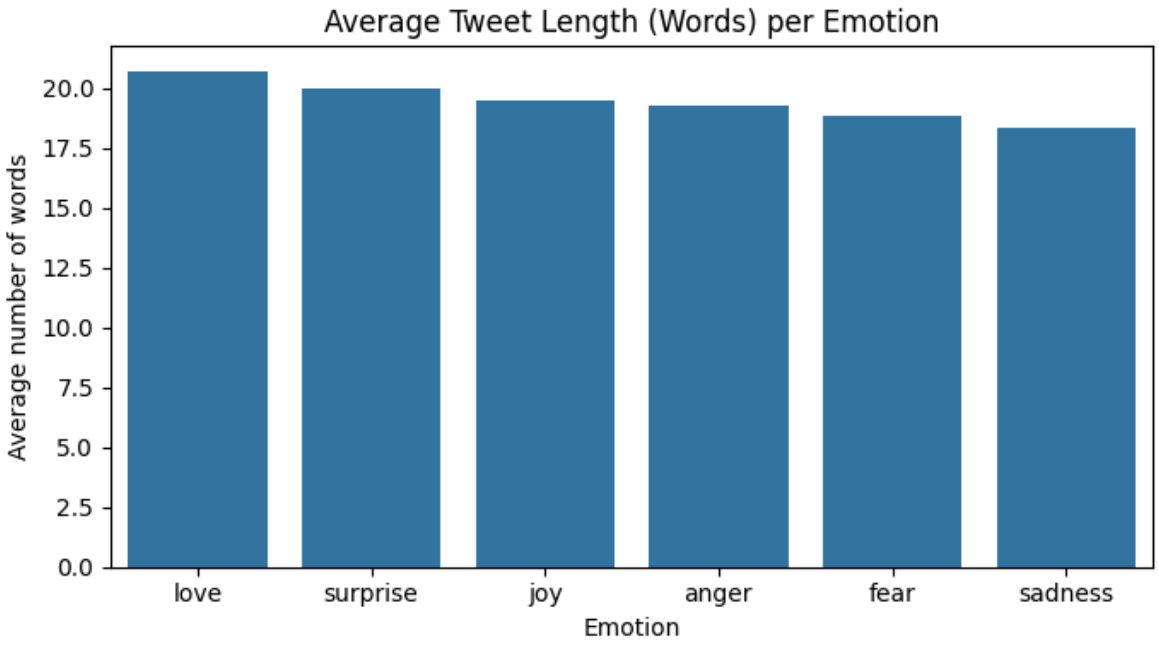
Average Tweet Length (in words) per Emotion

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

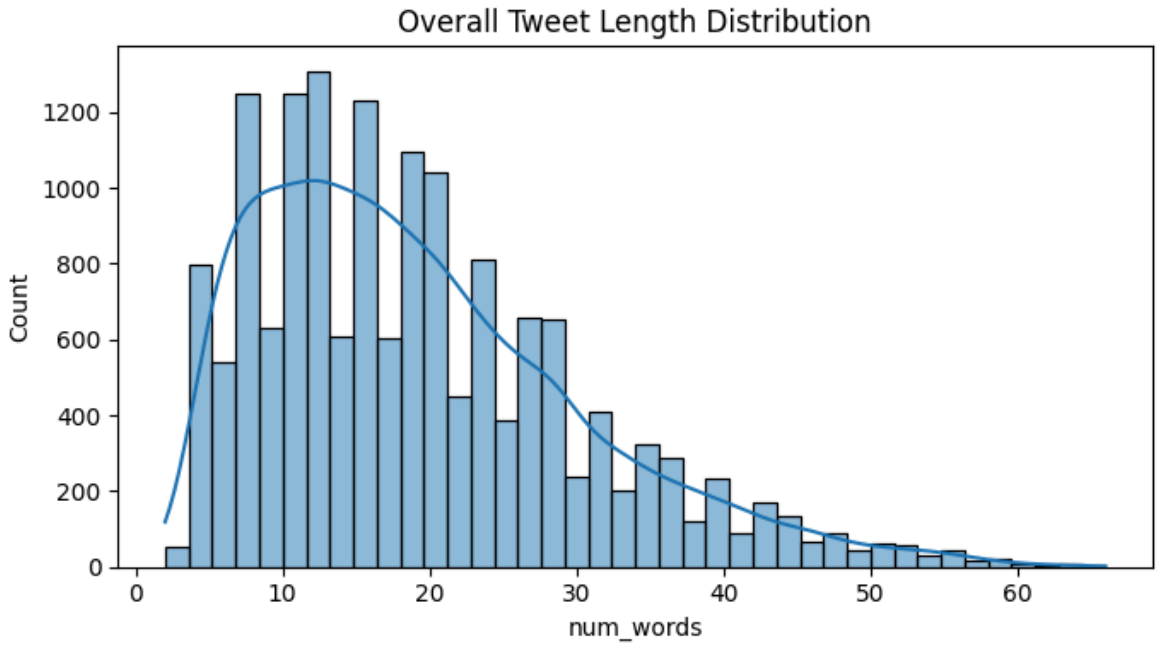
- **Love** tweets are the longest (~21 words), often containing more detailed emotional expression.
- **Surprise** and **Joy** follow (~19–20 words), usually describing events or reactions.
- **Anger** and **Fear** are slightly shorter (~19 words), often more direct.

- **Sadness** tweets are the shortest (~18 words), typically brief statements of feeling.

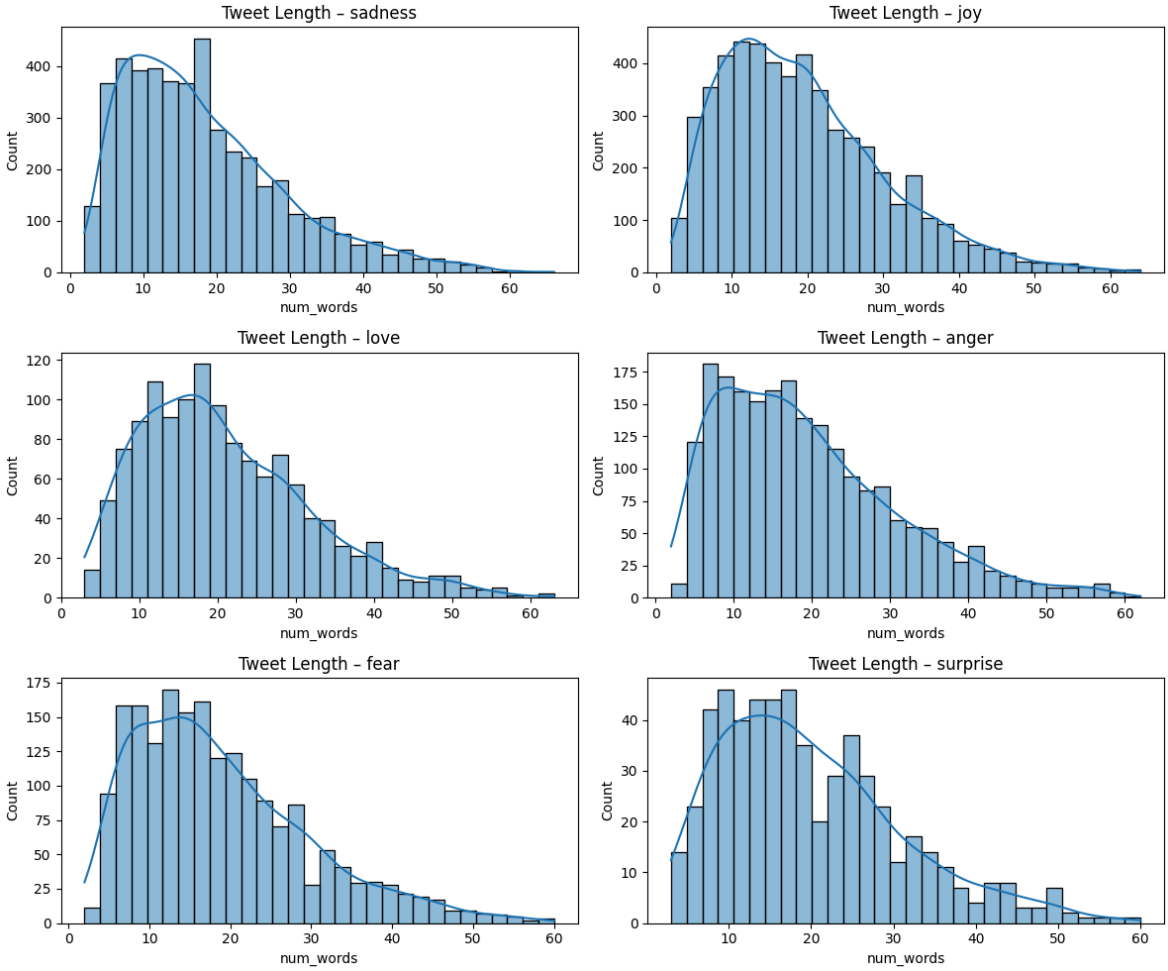
Overall, all emotions fall within a **tight range (18–21 words)**, confirming that tweets are consistently short and supporting a moderate max sequence length (e.g., 40–50 tokens) for modeling.

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, BiLSTM, and BERT all efficient to train.

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

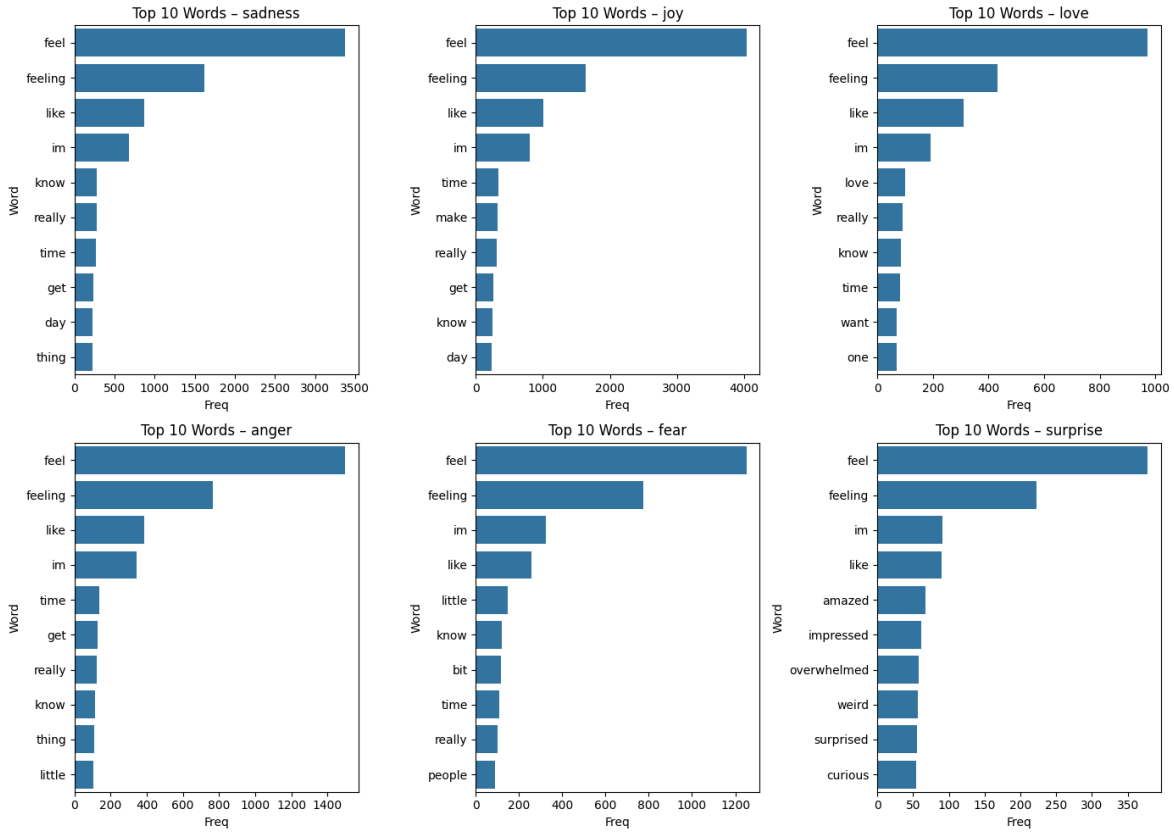
Quick Look — Per Emotion Label

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

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

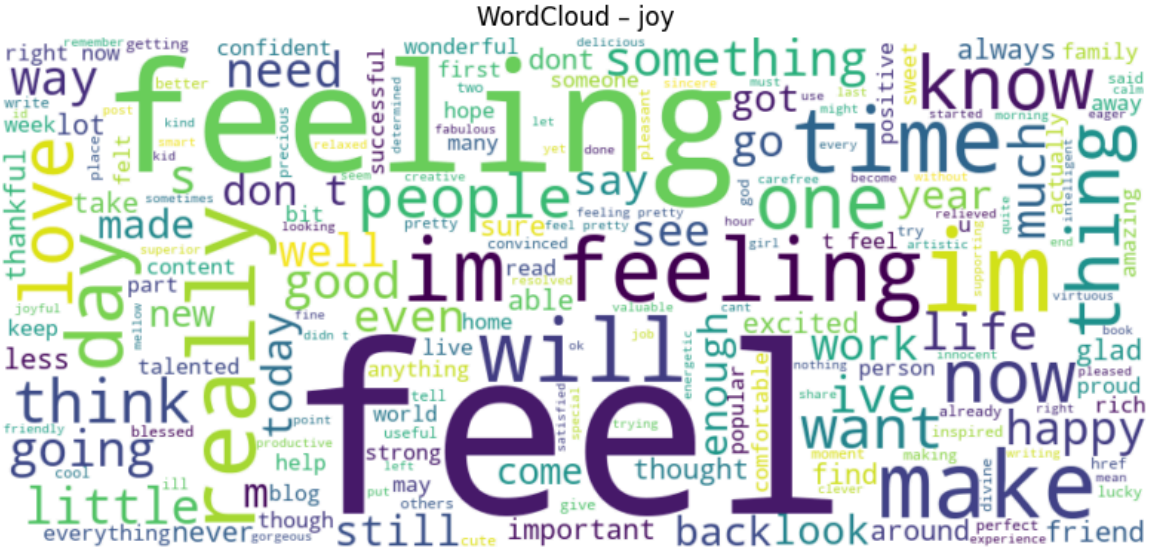
personal nature of Twitter content. These frequent terms provide an initial sense of common themes before emotion-specific analysis.

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}")
```

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}")
```



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]:
```
# 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)

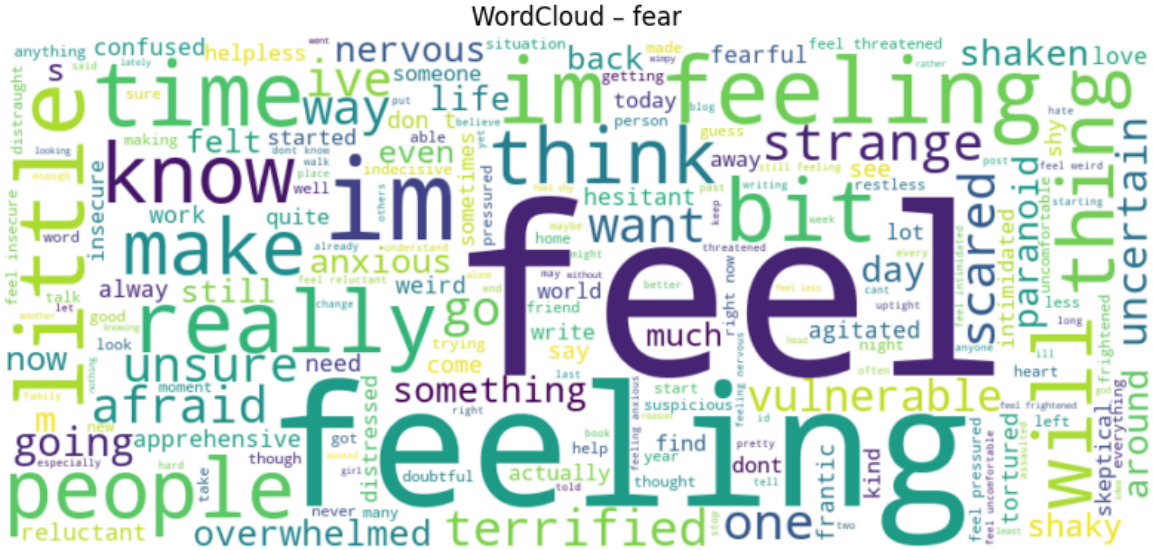
The love class features warm and affectionate words such as **love**, **lovely**, **caring**, **sweet**, **beloved**, and **romantic**. The vocabulary reflects themes of affection, connection, tenderness, and positive emotional bonding.

In [43]:
```
# 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}")
```

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.

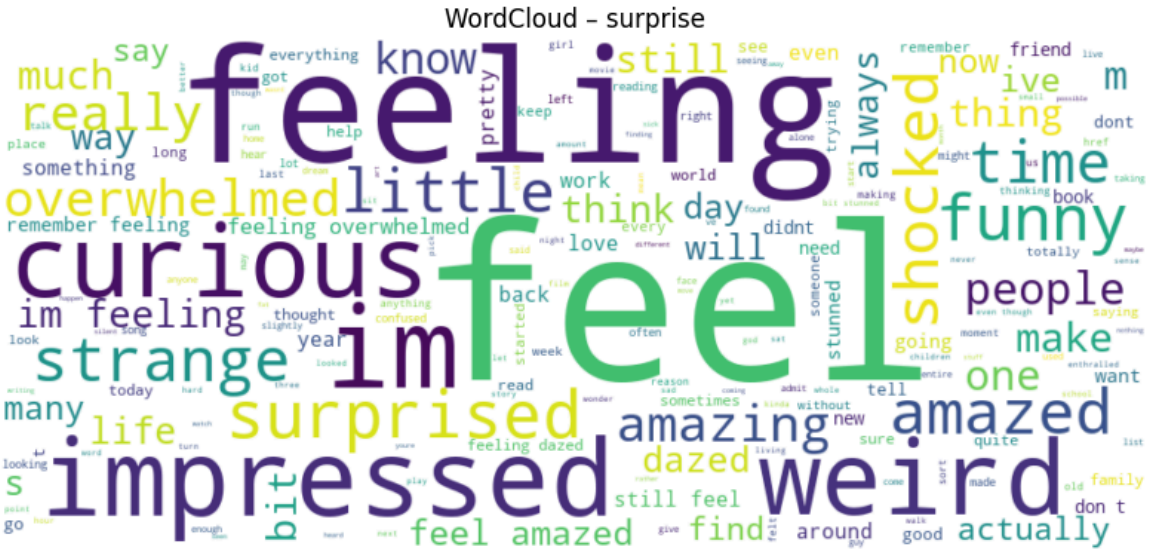
In [44]:
```
# fear class
lbl = 4
lbl_name = EMOTION_DICT[lbl]

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



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}")
```



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})
```

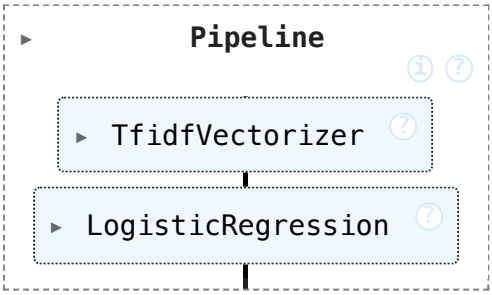
4. Model Training

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"))
```

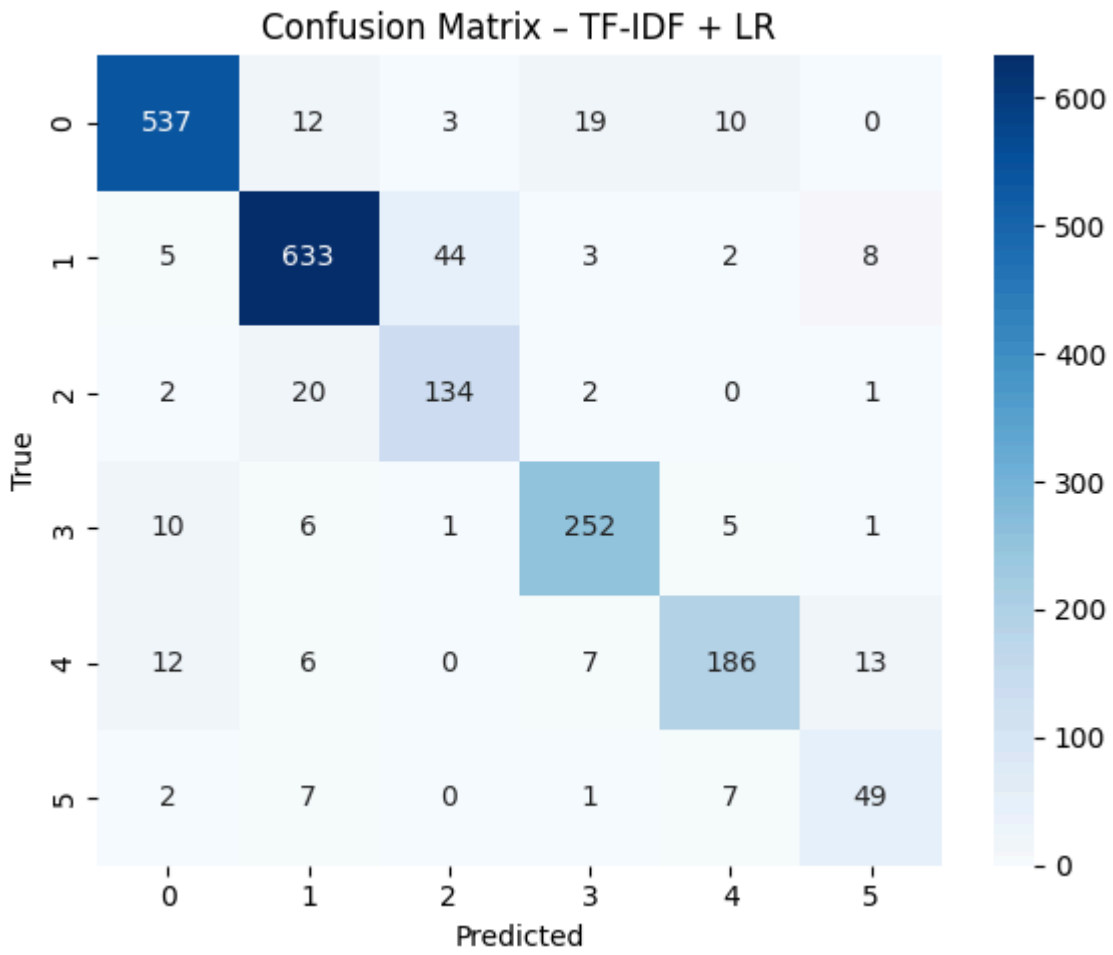
```
Result in TF-IDF + Logistic Regression:
Accuracy: 0.8955
Macro F1: 0.8512673279207412
Weighted F1: 0.8964496130401985
```

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**, showing strong performance across most emotion classes. High scores for **sadness**, **joy**, **anger**, and **fear** indicate that the model captures clear lexical cues associated with these emotions.

The **love** and **surprise** classes show comparatively lower F1-scores (0.79 and 0.71), which is expected due to their smaller dataset size and more subtle or overlapping linguistic patterns. Misclassifications in the confusion matrix reveal that *love* tweets are often predicted as *joy*, and *surprise* tweets sometimes overlap with *joy* or *fear*, suggesting semantic proximity between these emotional expressions.

Overall, the model offers a strong baseline with balanced macro and weighted averages (0.85 and 0.90), demonstrating its robustness despite class imbalance.

4.2 BiLSTM 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))
```

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

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]
```

BiLSTM model Training

In [67]:
```
BiLSTM_EPOCH = 150
```

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

inputs = layers.Input(shape=(MAX_SEQ_LEN,))
```

```
x = layers.Embedding(
    input_dim=vocab_size,
    output_dim=EMB_DIM,
    weights=[embedding_matrix],
    input_length=MAX_SEQ_LEN,
    trainable=False
)(inputs)
x = layers.Bidirectional(layers.LSTM(64))(x)
x = layers.Dropout(0.3)(x)
x = layers.Dense(128, activation="relu")(x)
outputs = layers.Dense(NUM_CLASSES, activation="softmax")(x)

bilstm = models.Model(inputs, outputs)
bilstm.compile(
    loss="sparse_categorical_crossentropy",
    optimizer=tf.keras.optimizers.Adam(2e-3),
    metrics=["accuracy"],
)

bilstm.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: "functional"

Layer (type)	Output Shape	P
input_layer (InputLayer)	(None, 40)	
embedding (Embedding)	(None, 40, 100)	1,5
bidirectional (Bidirectional)	(None, 128)	
dropout (Dropout)	(None, 128)	
dense (Dense)	(None, 128)	
dense_1 (Dense)	(None, 6)	

Total params: 1,612,966 (6.15 MB)

Trainable params: 101,766 (397.52 KB)

Non-trainable params: 1,511,200 (5.76 MB)

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]:
```
history = bilstm.fit(
    X_train_seq, y_train,
    validation_data=(X_dev_seq, y_dev),
    batch_size=64,
    epochs=BiLSTM_EPOCH,
    class_weight=class_weights,
    verbose=1
)

y_pred_bilstm = np.argmax(bilstm.predict(X_test_seq), axis=1)
```

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

```
Epoch 1/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.1821 - loss: 1.773
7 - val_accuracy: 0.1570 - val_loss: 1.7602
Epoch 2/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.2204 - loss: 1.734
0 - val_accuracy: 0.2195 - val_loss: 1.7125
Epoch 3/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.2433 - loss: 1.695
9 - val_accuracy: 0.2565 - val_loss: 1.6567
Epoch 4/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.2795 - loss: 1.652
4 - val_accuracy: 0.2810 - val_loss: 1.6316
Epoch 5/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.3176 - loss: 1.604
7 - val_accuracy: 0.3410 - val_loss: 1.5547
Epoch 6/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.3501 - loss: 1.554
9 - val_accuracy: 0.3770 - val_loss: 1.5034
Epoch 7/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.3677 - loss: 1.498
2 - val_accuracy: 0.3630 - val_loss: 1.5148
Epoch 8/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.3853 - loss: 1.453
0 - val_accuracy: 0.3845 - val_loss: 1.4562
Epoch 9/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.3986 - loss: 1.402
6 - val_accuracy: 0.4060 - val_loss: 1.3932
Epoch 10/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.4112 - loss: 1.352
1 - val_accuracy: 0.4280 - val_loss: 1.3480
Epoch 11/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.4238 - loss: 1.294
8 - val_accuracy: 0.4040 - val_loss: 1.3726
Epoch 12/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.4441 - loss: 1.250
7 - val_accuracy: 0.4825 - val_loss: 1.2481
Epoch 13/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.4629 - loss: 1.199
3 - val_accuracy: 0.4890 - val_loss: 1.2156
Epoch 14/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.4788 - loss: 1.158
0 - val_accuracy: 0.4575 - val_loss: 1.2550
Epoch 15/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.4962 - loss: 1.116
3 - val_accuracy: 0.5135 - val_loss: 1.1537
Epoch 16/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.5121 - loss: 1.065
8 - val_accuracy: 0.5340 - val_loss: 1.1270
Epoch 17/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.5344 - loss: 1.030
1 - val_accuracy: 0.5410 - val_loss: 1.0981
Epoch 18/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.5438 - loss: 1.009
6 - val_accuracy: 0.5665 - val_loss: 1.0637
Epoch 19/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.5551 - loss: 0.967
8 - val_accuracy: 0.5740 - val_loss: 1.0370
Epoch 20/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.5678 - loss: 0.949
7 - val_accuracy: 0.5915 - val_loss: 1.0094
```

```
Epoch 21/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.5778 - loss: 0.915
8 - val_accuracy: 0.6055 - val_loss: 0.9674
Epoch 22/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.5919 - loss: 0.890
4 - val_accuracy: 0.6100 - val_loss: 0.9723
Epoch 23/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6099 - loss: 0.852
7 - val_accuracy: 0.6225 - val_loss: 0.9501
Epoch 24/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6068 - loss: 0.847
1 - val_accuracy: 0.6290 - val_loss: 0.9183
Epoch 25/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.6174 - loss: 0.816
2 - val_accuracy: 0.6010 - val_loss: 0.9680
Epoch 26/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6360 - loss: 0.788
7 - val_accuracy: 0.6055 - val_loss: 0.9744
Epoch 27/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6384 - loss: 0.764
5 - val_accuracy: 0.6360 - val_loss: 0.9001
Epoch 28/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6318 - loss: 0.805
4 - val_accuracy: 0.6445 - val_loss: 0.8938
Epoch 29/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6530 - loss: 0.749
2 - val_accuracy: 0.6510 - val_loss: 0.8690
Epoch 30/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.6648 - loss: 0.718
9 - val_accuracy: 0.6720 - val_loss: 0.8380
Epoch 31/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6719 - loss: 0.687
8 - val_accuracy: 0.6510 - val_loss: 0.8854
Epoch 32/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6809 - loss: 0.683
9 - val_accuracy: 0.6615 - val_loss: 0.8445
Epoch 33/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6867 - loss: 0.657
8 - val_accuracy: 0.6460 - val_loss: 0.8922
Epoch 34/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6867 - loss: 0.668
3 - val_accuracy: 0.6735 - val_loss: 0.8520
Epoch 35/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.7051 - loss: 0.636
5 - val_accuracy: 0.6890 - val_loss: 0.8182
Epoch 36/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.6996 - loss: 0.634
0 - val_accuracy: 0.6900 - val_loss: 0.8313
Epoch 37/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7192 - loss: 0.594
7 - val_accuracy: 0.6815 - val_loss: 0.7936
Epoch 38/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7107 - loss: 0.598
1 - val_accuracy: 0.7050 - val_loss: 0.7635
Epoch 39/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7246 - loss: 0.582
9 - val_accuracy: 0.7145 - val_loss: 0.7822
Epoch 40/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7278 - loss: 0.585
6 - val_accuracy: 0.6920 - val_loss: 0.8254
```

```
Epoch 41/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7304 - loss: 0.562
1 - val_accuracy: 0.6970 - val_loss: 0.7836
Epoch 42/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7401 - loss: 0.544
9 - val_accuracy: 0.7225 - val_loss: 0.7537
Epoch 43/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.7487 - loss: 0.528
7 - val_accuracy: 0.7040 - val_loss: 0.7892
Epoch 44/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7483 - loss: 0.537
5 - val_accuracy: 0.7215 - val_loss: 0.7401
Epoch 45/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7415 - loss: 0.558
2 - val_accuracy: 0.7335 - val_loss: 0.7339
Epoch 46/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.7579 - loss: 0.511
9 - val_accuracy: 0.7465 - val_loss: 0.7047
Epoch 47/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7589 - loss: 0.513
4 - val_accuracy: 0.7170 - val_loss: 0.7617
Epoch 48/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7686 - loss: 0.487
3 - val_accuracy: 0.7325 - val_loss: 0.7399
Epoch 49/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7751 - loss: 0.464
9 - val_accuracy: 0.7290 - val_loss: 0.7468
Epoch 50/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7467 - loss: 0.541
2 - val_accuracy: 0.7455 - val_loss: 0.7109
Epoch 51/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7722 - loss: 0.487
3 - val_accuracy: 0.7450 - val_loss: 0.7168
Epoch 52/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7743 - loss: 0.475
8 - val_accuracy: 0.7440 - val_loss: 0.7203
Epoch 53/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.7913 - loss: 0.439
5 - val_accuracy: 0.7365 - val_loss: 0.7262
Epoch 54/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7868 - loss: 0.444
7 - val_accuracy: 0.7630 - val_loss: 0.6900
Epoch 55/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7954 - loss: 0.429
3 - val_accuracy: 0.7460 - val_loss: 0.7308
Epoch 56/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7972 - loss: 0.430
5 - val_accuracy: 0.7655 - val_loss: 0.6987
Epoch 57/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7933 - loss: 0.435
8 - val_accuracy: 0.7580 - val_loss: 0.7053
Epoch 58/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8019 - loss: 0.417
0 - val_accuracy: 0.7620 - val_loss: 0.7213
Epoch 59/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8035 - loss: 0.410
4 - val_accuracy: 0.7480 - val_loss: 0.7203
Epoch 60/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8116 - loss: 0.398
6 - val_accuracy: 0.7630 - val_loss: 0.7168
```

```
Epoch 61/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8050 - loss: 0.414
9 - val_accuracy: 0.7580 - val_loss: 0.7658
Epoch 62/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8085 - loss: 0.408
8 - val_accuracy: 0.7770 - val_loss: 0.6730
Epoch 63/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8179 - loss: 0.384
5 - val_accuracy: 0.7680 - val_loss: 0.7062
Epoch 64/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8173 - loss: 0.382
0 - val_accuracy: 0.7555 - val_loss: 0.7080
Epoch 65/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8173 - loss: 0.390
1 - val_accuracy: 0.7620 - val_loss: 0.6958
Epoch 66/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7735 - loss: 0.505
3 - val_accuracy: 0.7620 - val_loss: 0.6750
Epoch 67/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8227 - loss: 0.376
9 - val_accuracy: 0.7645 - val_loss: 0.7223
Epoch 68/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8312 - loss: 0.350
8 - val_accuracy: 0.7730 - val_loss: 0.7371
Epoch 69/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8168 - loss: 0.385
7 - val_accuracy: 0.7710 - val_loss: 0.7280
Epoch 70/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 19ms/step - accuracy: 0.8221 - loss: 0.372
8 - val_accuracy: 0.7500 - val_loss: 0.8000
Epoch 71/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.8203 - loss: 0.370
6 - val_accuracy: 0.7810 - val_loss: 0.7019
Epoch 72/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8244 - loss: 0.356
6 - val_accuracy: 0.7705 - val_loss: 0.6796
Epoch 73/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8273 - loss: 0.360
1 - val_accuracy: 0.7770 - val_loss: 0.6670
Epoch 74/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8376 - loss: 0.342
7 - val_accuracy: 0.7805 - val_loss: 0.7038
Epoch 75/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8292 - loss: 0.356
6 - val_accuracy: 0.7710 - val_loss: 0.7391
Epoch 76/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8269 - loss: 0.368
8 - val_accuracy: 0.7820 - val_loss: 0.7017
Epoch 77/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8424 - loss: 0.330
0 - val_accuracy: 0.7815 - val_loss: 0.7353
Epoch 78/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8343 - loss: 0.362
2 - val_accuracy: 0.7800 - val_loss: 0.7397
Epoch 79/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8413 - loss: 0.327
8 - val_accuracy: 0.7905 - val_loss: 0.6930
Epoch 80/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8513 - loss: 0.314
3 - val_accuracy: 0.7930 - val_loss: 0.6867
```

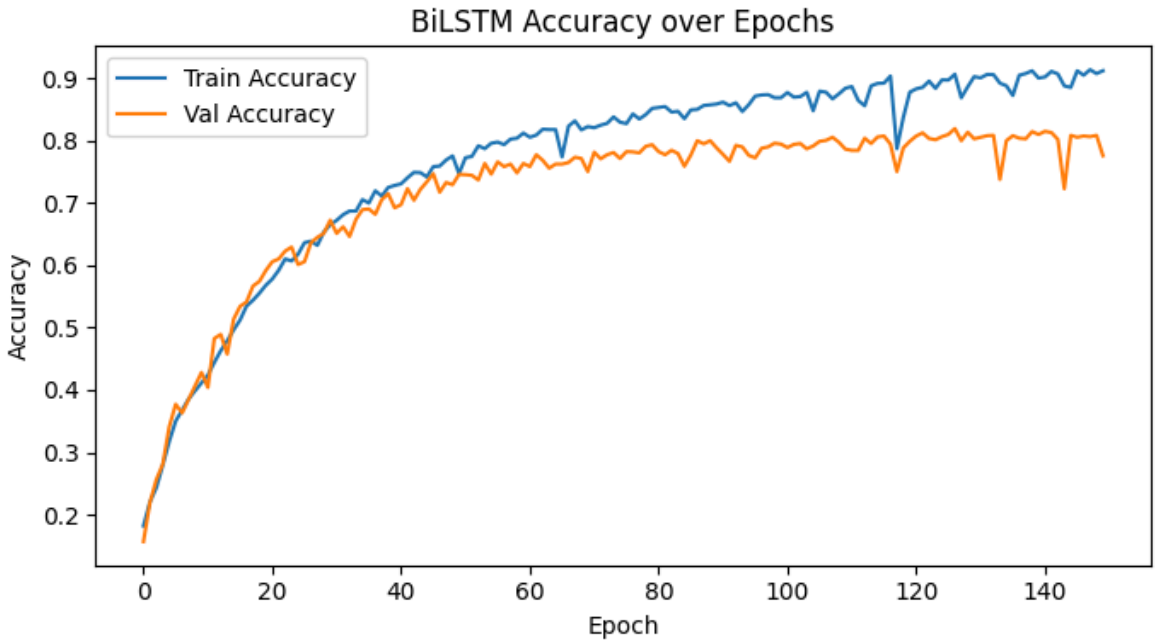
```
Epoch 81/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8529 - loss: 0.309
1 - val_accuracy: 0.7820 - val_loss: 0.6885
Epoch 82/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.8542 - loss: 0.309
0 - val_accuracy: 0.7770 - val_loss: 0.7026
Epoch 83/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.8454 - loss: 0.331
6 - val_accuracy: 0.7840 - val_loss: 0.7129
Epoch 84/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8466 - loss: 0.328
4 - val_accuracy: 0.7790 - val_loss: 0.7318
Epoch 85/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8347 - loss: 0.347
6 - val_accuracy: 0.7585 - val_loss: 0.7860
Epoch 86/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8488 - loss: 0.329
8 - val_accuracy: 0.7765 - val_loss: 0.7144
Epoch 87/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8497 - loss: 0.321
5 - val_accuracy: 0.7995 - val_loss: 0.6559
Epoch 88/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.8559 - loss: 0.303
3 - val_accuracy: 0.7945 - val_loss: 0.6998
Epoch 89/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8569 - loss: 0.300
4 - val_accuracy: 0.7995 - val_loss: 0.6388
Epoch 90/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8583 - loss: 0.301
3 - val_accuracy: 0.7880 - val_loss: 0.7225
Epoch 91/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8612 - loss: 0.291
3 - val_accuracy: 0.7775 - val_loss: 0.7230
Epoch 92/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8558 - loss: 0.311
8 - val_accuracy: 0.7665 - val_loss: 0.7735
Epoch 93/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8601 - loss: 0.288
3 - val_accuracy: 0.7920 - val_loss: 0.7308
Epoch 94/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8463 - loss: 0.327
4 - val_accuracy: 0.7895 - val_loss: 0.7222
Epoch 95/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8573 - loss: 0.310
0 - val_accuracy: 0.7760 - val_loss: 0.7511
Epoch 96/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8710 - loss: 0.272
8 - val_accuracy: 0.7720 - val_loss: 0.7736
Epoch 97/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8728 - loss: 0.269
6 - val_accuracy: 0.7875 - val_loss: 0.7964
Epoch 98/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8733 - loss: 0.263
2 - val_accuracy: 0.7895 - val_loss: 0.7403
Epoch 99/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8684 - loss: 0.275
8 - val_accuracy: 0.7955 - val_loss: 0.7176
Epoch 100/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8684 - loss: 0.282
8 - val_accuracy: 0.7940 - val_loss: 0.7104
```

```
Epoch 101/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8766 - loss: 0.261
2 - val_accuracy: 0.7885 - val_loss: 0.7313
Epoch 102/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8693 - loss: 0.280
5 - val_accuracy: 0.7935 - val_loss: 0.7704
Epoch 103/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8702 - loss: 0.266
9 - val_accuracy: 0.7950 - val_loss: 0.7651
Epoch 104/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8769 - loss: 0.262
0 - val_accuracy: 0.7865 - val_loss: 0.7340
Epoch 105/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8476 - loss: 0.338
3 - val_accuracy: 0.7910 - val_loss: 0.7250
Epoch 106/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8788 - loss: 0.262
8 - val_accuracy: 0.7985 - val_loss: 0.7310
Epoch 107/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8772 - loss: 0.263
6 - val_accuracy: 0.8000 - val_loss: 0.7386
Epoch 108/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8666 - loss: 0.290
9 - val_accuracy: 0.8050 - val_loss: 0.7283
Epoch 109/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.8746 - loss: 0.268
3 - val_accuracy: 0.7970 - val_loss: 0.7381
Epoch 110/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8836 - loss: 0.243
1 - val_accuracy: 0.7860 - val_loss: 0.7742
Epoch 111/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8863 - loss: 0.243
5 - val_accuracy: 0.7840 - val_loss: 0.8130
Epoch 112/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8639 - loss: 0.296
1 - val_accuracy: 0.7840 - val_loss: 0.7843
Epoch 113/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8555 - loss: 0.324
4 - val_accuracy: 0.8035 - val_loss: 0.6872
Epoch 114/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8874 - loss: 0.236
3 - val_accuracy: 0.7950 - val_loss: 0.7652
Epoch 115/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8913 - loss: 0.223
7 - val_accuracy: 0.8055 - val_loss: 0.7995
Epoch 116/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8917 - loss: 0.231
0 - val_accuracy: 0.8070 - val_loss: 0.7962
Epoch 117/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.9031 - loss: 0.210
3 - val_accuracy: 0.7940 - val_loss: 0.7843
Epoch 118/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.7868 - loss: 0.571
4 - val_accuracy: 0.7500 - val_loss: 0.6963
Epoch 119/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8369 - loss: 0.359
3 - val_accuracy: 0.7880 - val_loss: 0.6981
Epoch 120/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8773 - loss: 0.256
6 - val_accuracy: 0.7985 - val_loss: 0.7300
```

```
Epoch 121/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8826 - loss: 0.249
0 - val_accuracy: 0.8070 - val_loss: 0.7272
Epoch 122/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8851 - loss: 0.240
0 - val_accuracy: 0.8120 - val_loss: 0.7255
Epoch 123/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8950 - loss: 0.221
6 - val_accuracy: 0.8030 - val_loss: 0.7589
Epoch 124/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8834 - loss: 0.257
7 - val_accuracy: 0.8010 - val_loss: 0.7177
Epoch 125/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8969 - loss: 0.219
6 - val_accuracy: 0.8060 - val_loss: 0.7584
Epoch 126/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8967 - loss: 0.215
3 - val_accuracy: 0.8095 - val_loss: 0.7746
Epoch 127/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.9060 - loss: 0.199
2 - val_accuracy: 0.8190 - val_loss: 0.7790
Epoch 128/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8683 - loss: 0.309
0 - val_accuracy: 0.7995 - val_loss: 0.6894
Epoch 129/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8859 - loss: 0.253
2 - val_accuracy: 0.8130 - val_loss: 0.7215
Epoch 130/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.9021 - loss: 0.203
8 - val_accuracy: 0.8025 - val_loss: 0.7492
Epoch 131/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.9003 - loss: 0.207
8 - val_accuracy: 0.8050 - val_loss: 0.7103
Epoch 132/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.9056 - loss: 0.195
4 - val_accuracy: 0.8075 - val_loss: 0.7698
Epoch 133/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.9055 - loss: 0.201
9 - val_accuracy: 0.8080 - val_loss: 0.7799
Epoch 134/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8917 - loss: 0.246
7 - val_accuracy: 0.7375 - val_loss: 0.9025
Epoch 135/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.8875 - loss: 0.235
6 - val_accuracy: 0.7995 - val_loss: 0.7983
Epoch 136/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.8719 - loss: 0.285
0 - val_accuracy: 0.8075 - val_loss: 0.7259
Epoch 137/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.9043 - loss: 0.206
6 - val_accuracy: 0.8035 - val_loss: 0.7787
Epoch 138/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 17ms/step - accuracy: 0.9072 - loss: 0.195
5 - val_accuracy: 0.8025 - val_loss: 0.7702
Epoch 139/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 18ms/step - accuracy: 0.9115 - loss: 0.184
1 - val_accuracy: 0.8140 - val_loss: 0.7804
Epoch 140/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 6s 25ms/step - accuracy: 0.8999 - loss: 0.214
4 - val_accuracy: 0.8095 - val_loss: 0.7159
```

```
Epoch 141/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 18ms/step - accuracy: 0.9016 - loss: 0.222
4 - val_accuracy: 0.8145 - val_loss: 0.7570
Epoch 142/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.9106 - loss: 0.195
7 - val_accuracy: 0.8125 - val_loss: 0.8125
Epoch 143/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 6s 22ms/step - accuracy: 0.9064 - loss: 0.197
5 - val_accuracy: 0.8015 - val_loss: 0.7677
Epoch 144/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 19ms/step - accuracy: 0.8873 - loss: 0.253
2 - val_accuracy: 0.7225 - val_loss: 0.9388
Epoch 145/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 19ms/step - accuracy: 0.8849 - loss: 0.247
7 - val_accuracy: 0.8080 - val_loss: 0.7672
Epoch 146/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 18ms/step - accuracy: 0.9115 - loss: 0.189
1 - val_accuracy: 0.8050 - val_loss: 0.7742
Epoch 147/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 19ms/step - accuracy: 0.9045 - loss: 0.208
2 - val_accuracy: 0.8070 - val_loss: 0.7351
Epoch 148/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.9133 - loss: 0.185
3 - val_accuracy: 0.8060 - val_loss: 0.7438
Epoch 149/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.9071 - loss: 0.205
1 - val_accuracy: 0.8080 - val_loss: 0.7980
Epoch 150/150
250/250 ━━━━━━━━━━━━━━━━━━━━ 5s 18ms/step - accuracy: 0.9116 - loss: 0.192
2 - val_accuracy: 0.7755 - val_loss: 0.9431
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
Results in Word2Vec + BiLSTM:
Accuracy : 0.784
Macro F1 : 0.7363371687706803
Weighted : 0.7848763169477218
```

In [71]:
```
# BiLSTM Training Curve (Accuracy)
plt.figure(figsize=(7,4))
plt.plot(history.history["accuracy"], label="Train Accuracy")
plt.plot(history.history["val_accuracy"], label="Val Accuracy")
plt.title("BiLSTM Accuracy over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.tight_layout()
plt.show()
```



Quick Look — BiLSTM Accuracy over Epochs

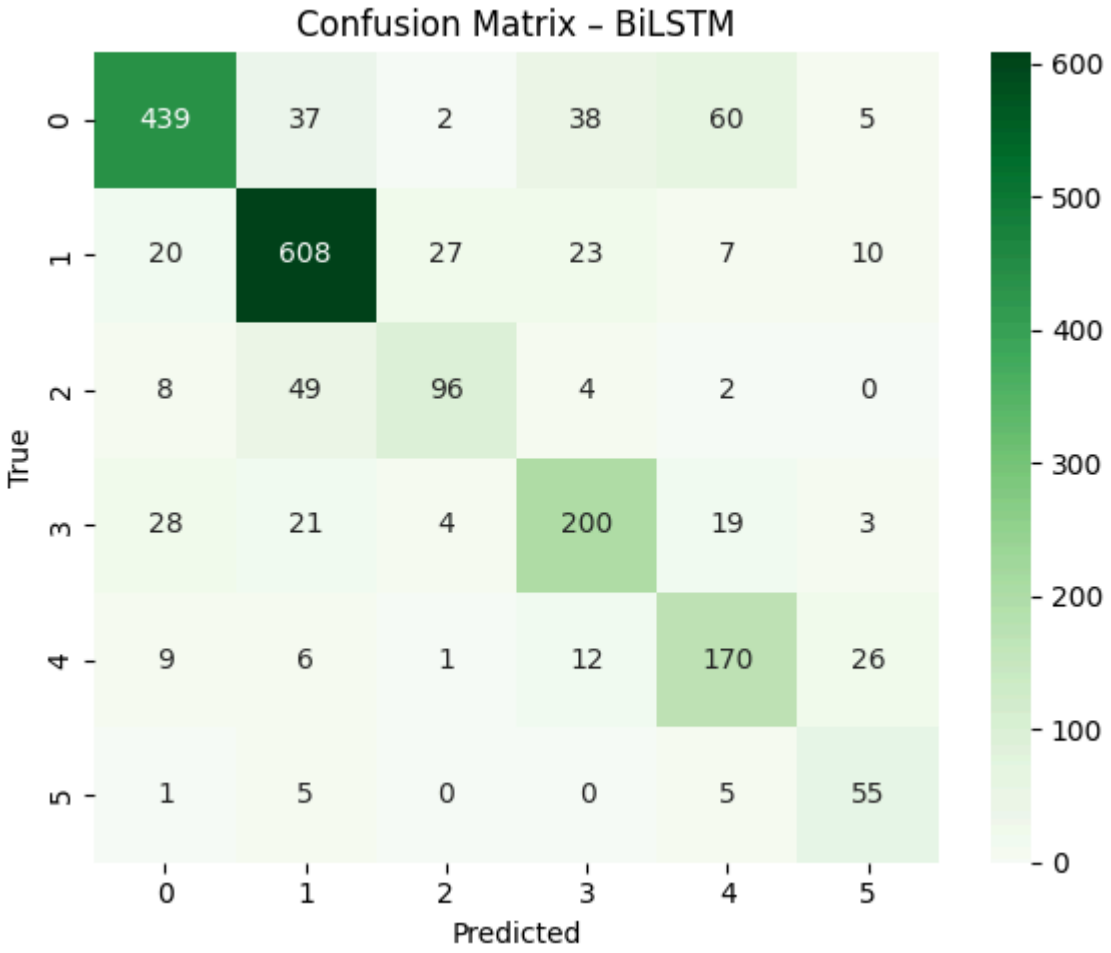
- Training accuracy increases steadily and reaches ~0.91, indicating effective learning.
- Validation accuracy improves rapidly early on, then plateaus around ~0.78–0.81.
- Best generalization appears before the final epochs → early stopping could improve efficiency.

In [72]:
```
print("\nClassification Report:\n")
print(
    classification_report(
        y_test,
        y_pred_bilstm,
        target_names=EMOTION_DICT.values()
    )
)
```

```
Classification Report:

              precision    recall  f1-score   support

     sadness       0.87      0.76      0.81       581
         joy       0.84      0.87      0.86       695
        love       0.74      0.60      0.66       159
       anger       0.72      0.73      0.72       275
        fear       0.65      0.76      0.70       224
    surprise       0.56      0.83      0.67        66

    accuracy                           0.78      2000
   macro avg       0.73      0.76      0.74      2000
weighted avg       0.79      0.78      0.78      2000
```

In [73]:
```
cm_bilstm = confusion_matrix(y_test, y_pred_bilstm)
plt.figure(figsize=(6,5))
sns.heatmap(cm_bilstm, annot=True, fmt="d", cmap="Greens",
            xticklabels=label_encoder.classes_,
            yticklabels=label_encoder.classes_)
```

```
plt.title("Confusion Matrix – BiLSTM")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.show()
```



Performance Summary (BiLSTM)

The BiLSTM model achieves **78% overall accuracy**, which is noticeably lower than the TF-IDF + Logistic Regression baseline. While the model performs well on **joy** (F1 = 0.86) and reasonably on **sadness, anger, and fear**, it struggles with **love** and **surprise**, which obtain the lowest F1-scores (0.66 and 0.67).

The confusion matrix reveals substantial overlap between semantically related emotions. In particular, **love** is frequently misclassified as **joy**, while **surprise** is often confused with **joy** or **fear**, indicating difficulty in distinguishing subtle emotional cues.

Overall, the BiLSTM provides moderate performance but is more sensitive to class imbalance and limited context in short tweets. Compared to the TF-IDF baseline, it requires stronger embeddings, more data, or regularization (e.g., early stopping) to generalize effectively.

4.3 BERT dataset

BERT dataset

In [74]:
```
BERT_MODEL = "bert-base-uncased"
bert_tokenizer = AutoTokenizer.from_pretrained(BERT_MODEL)
```

In [75]:
```
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 [76]:
```
BERT_EPOCH=5
```

In [77]:
```
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 [78]:
```
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 [79]:
```
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 [80]:
```
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 39:02, Epoch 5/5]

Step	Training Loss
50	1.656000
100	1.384200
150	1.174400
200	0.920400
250	0.695900
300	0.489300
350	0.433600
400	0.413700
450	0.335600
500	0.290000
550	0.273700
600	0.249500
650	0.282800
700	0.212600
750	0.230300
800	0.237700
850	0.244800
900	0.211800
950	0.237900
1000	0.241700
1050	0.127200
1100	0.122900
1150	0.182700
1200	0.149600
1250	0.156800
1300	0.154300
1350	0.147700
1400	0.156600
1450	0.123900
1500	0.202300
1550	0.112900
1600	0.151300
1650	0.179000

Step	Training Loss
1700	0.161700
1750	0.169600
1800	0.143800
1850	0.185900
1900	0.126500
1950	0.111900
2000	0.143500
2050	0.122400
2100	0.098100
2150	0.101800
2200	0.098700
2250	0.088000
2300	0.116000
2350	0.095700
2400	0.118000
2450	0.140700
2500	0.104100
2550	0.101400
2600	0.086100
2650	0.103500
2700	0.132600
2750	0.077200
2800	0.080300
2850	0.115500
2900	0.108600
2950	0.093300
3000	0.090300
3050	0.058100
3100	0.079500
3150	0.065500
3200	0.092600
3250	0.041700
3300	0.055900
3350	0.076700

Step	Training Loss
3400	0.134200
3450	0.069100
3500	0.066800
3550	0.081900
3600	0.073700
3650	0.085000
3700	0.113600
3750	0.077400
3800	0.067300
3850	0.065000
3900	0.088500
3950	0.068000
4000	0.068800
4050	0.047100
4100	0.046200
4150	0.048400
4200	0.042100
4250	0.045700
4300	0.029800
4350	0.068500
4400	0.042200
4450	0.045800
4500	0.040800
4550	0.052900
4600	0.044200
4650	0.080100
4700	0.051500
4750	0.054300
4800	0.048900
4850	0.074400
4900	0.048200
4950	0.047400
5000	0.045700

```
/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[80]:
```
TrainOutput(global_step=5000, training_loss=0.1783176291704178, metrics=
{'train_runtime': 2343.7365, 'train_samples_per_second': 34.134, 'train_
steps_per_second': 2.133, 'total_flos': 2631205048320000.0, 'train_los
s': 0.1783176291704178, 'epoch': 5.0})
```

In [81]:
```
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 [82]:
```
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 : 0.925
Macro F1 : 0.8783947061429723
Weighted : 0.9247865128496794
```

In [83]:
```
print(
    classification_report(
        y_test,
        y_pred_bert,
        target_names=EMOTION_DICT.values()
    )
)
```

```
                precision    recall  f1-score   support

       sadness       0.96      0.98      0.97       581
           joy       0.95      0.94      0.94       695
          love       0.81      0.80      0.80       159
         anger       0.94      0.91      0.92       275
          fear       0.88      0.90      0.89       224
      surprise       0.75      0.73      0.74        66

      accuracy                           0.93      2000
     macro avg       0.88      0.88      0.88      2000
  weighted avg       0.92      0.93      0.92      2000
```

Performance Summary (BERT Fine-Tuning)

The fine-tuned BERT model delivers the strongest performance among all tested approaches, achieving **≈93% overall accuracy** with high precision and recall across most emotion categories. Major classes such as **sadness, joy, anger, and fear** show excellent F1-scores (≥ 0.89), indicating that BERT effectively captures contextual and semantic cues in tweets.

Although **love** and **surprise** remain the most challenging classes, their F1-scores (**0.80 and 0.74**, respectively) still outperform both the TF-IDF + Logistic Regression and BiLSTM models by a clear margin. The improved **macro-F1 (0.88)** demonstrates better balance across all emotion categories despite dataset imbalance.

Overall, BERT provides a substantial performance boost, showing strong generalization and robust emotion discrimination even for short, informal text.

5. Result and Conclusion

5.1 Model Evaluation

In [84]:
```
results = []
```

In [85]:
```
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 [86]:
```
results.append({
        "Model": "Word2Vec + BiLSTM",
        "Accuracy": accuracy_score(y_test, y_pred_bilstm),
        "Macro F1": f1_score(y_test, y_pred_bilstm, average="macro"),
        "Weighted F1": f1_score(y_test, y_pred_bilstm, average="weighted"
    })
```

In [87]:
```
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 [88]:
```
pd.DataFrame(results)
```

Out[88]:

	Model	Accuracy	Macro F1	Weighted F1
0	TF-IDF + LR	0.8955	0.851267	0.896450
1	Word2Vec + BiLSTM	0.7840	0.736337	0.784876
2	BERT Fine-Tuning	0.9250	0.878395	0.924787

5.1 Conclusion & Recommendation

Conclusion

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

Model Comparison and Key Findings

- **TF-IDF + Logistic Regression**
  - Achieved strong baseline results with **89.6% accuracy** and **macro F1 = 0.85**
  - Demonstrated stable and reliable performance on high-frequency emotions such as *sadness*, *joy*, *anger*, and *fear*
  - Performance on minority classes (*love* and *surprise*) remained weaker, reflecting limited ability to capture nuanced semantic relationships
  - Results confirm that classical models with well-designed text preprocessing and feature extraction can remain highly competitive for short-text classification tasks
- **Word2Vec + BiLSTM**
  - Delivered noticeably lower performance (**78.4% accuracy**, **macro F1 = 0.74**)
  - Despite being a more expressive neural architecture, it failed to surpass the TF-IDF baseline
  - Training dynamics and evaluation suggest sensitivity to class imbalance, limited training data, and the short, informal nature of tweets
  - Highlights a common limitation of sequence-based neural models: without large-scale data and rich contextual signals, they may underperform simpler methods

- **BERT Fine-Tuning**
  - Achieved the strongest results by a clear margin:
    - **Accuracy: 92.5%**
    - **Macro F1: 0.88**
    - **Weighted F1: 0.92**
  - Exhibited consistently high precision and recall across all major emotion categories
  - Showed meaningful improvements for minority classes compared to TF-IDF and BiLSTM
  - Effectively leveraged pre-trained contextual representations to capture subtle emotional cues, idiomatic expressions, and semantic overlap common in social media text

Overall Observations

- There is a clear performance progression from **bag-of-words → sequential embeddings → contextual transformers**
- Models that incorporate **pre-trained contextual knowledge** generalize better to short, noisy, and informal language
- Class imbalance remains a challenge across all approaches, but **BERT demonstrates the highest robustness** under these conditions

Final Conclusion

Overall, the experiments confirm that **transformer-based architectures are significantly better suited for emotion classification in social media contexts**. While classical models provide strong and efficient baselines, and BiLSTM offers moderate gains in representation capacity, **BERT Fine-Tuning delivers the most reliable and scalable solution**, making it the best candidate for real-world deployment.

Recommendation

- Adopt **BERT Fine-Tuning** as the primary model for production use
- To further improve performance and robustness:
  - Apply class-aware loss functions (e.g., focal loss, class-balanced loss)
  - Experiment with domain-specific transformers such as **BERTweet** or **RoBERTa**
  - Incorporate additional signals like emojis, hashtags, or metadata alongside text
  - Optimize inference via model distillation or quantization
  - Periodically retrain the model 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 [ ]: