Abstraction：

In this Kaggle sentiment analysis task, I initially chose two approaches:

1. Using traditional NLP methods
2. Using pre-trained models for prediction

First, I tried the first method, which included data preprocessing, tokenization, lemmatization, and removal of stop words. Subsequently, I used external packages to determine the positive and negative polarity of each word and calculated the relationship between positive and negative word counts in each comment. After completing this and performing a series of statistical analyses and visualizations, I found it difficult to identify key factors influencing the sentiment analysis.

Later, I attempted the second method using BERTweet, which is a pre-trained model specifically designed for Twitter analysis. Since it already understands Twitter's context and special relationships, it doesn't require extensive preprocessing.

Finally, I decided to use BERTweet as my final tool.

Method 1：

data preprocessing

```
def clean_at(text):
    text = re.sub(r'@\w+', '', text)
    return text

def to_lower(text):
    text = text.lower()
    return text

def clean_LH(text):
    text = text.replace("<lh>", '')
    return text

def remove_url(text):
    # 使用正則表達式匹配 www. 開頭和 .com 結尾的網址
    # 這個正則表達式會匹配以 "www." 開頭且以 ".com" 結尾的網址
    text = re.sub(r'http[s]?://(?:www\.)?[\w-]+\.[a-z]{2,}/?', '', text)
    return text

def handle_hashtag(text, hashtags):
    if "#" in text:
        found_hashtags = re.findall(r'#\w+', text)

        for hashtag in found_hashtags:
            clean_hashtag = hashtag.lstrip('#')

            if clean_hashtag not in hashtags:
```

```python
                hashtags.append(clean_hashtag)

    return hashtags

# 將處理函數移到最外層
def process_single_row(args):
    """
    處理單行數據的函數
    """
    row, text_column, hashtags_column, pattern = args
    text = str(row[text_column])
    current_hashtags = row[hashtags_column]

    # 確保 current_hashtags 是列表
    if not isinstance(current_hashtags, list):
        current_hashtags = []

    if '#' in text:
        # 找出所有 hashtags
        found_tags = pattern.findall(text)
        # 移除 # 符號並轉換為集合
        new_tags = {tag.lstrip('#') for tag in found_tags}
        # 現有的 hashtags 轉換為集合
        current_tags = set(current_hashtags)
        # 合併並轉回列表
        return list(current_tags.union(new_tags))

    return current_hashtags

def process_chunk(args):
    chunk, text_column, hashtags_column, pattern = args
    result = []
    for _, row in chunk.iterrows():
        result.append(process_single_row((row, text_column, hashtags_column,
pattern)))
    return result

def process_hashtags_fast_v2(df, text_column='text',
hashtags_column='hashtags', batch_size=10000):
    """
    快速處理大量推文中的 hashtags，使用批次處理而不是多進程

    Parameters:
        df: 輸入的 DataFrame
        text_column: 包含文本的列名
        hashtags_column: 包含現有 hashtags 的列名
        batch_size: 每批處理的行數
```

```python
    Returns:
        處理後的 hashtags Series
    """
    # 預編譯正則表達式
    pattern = re.compile(r'#\w+')

    # 初始化結果列表
    results = []

    # 批次處理數據
    for start_idx in range(0, len(df), batch_size):
        end_idx = min(start_idx + batch_size, len(df))
        batch = df.iloc[start_idx:end_idx]

        # 處理當前批次
        batch_results = process_chunk((batch, text_column, hashtags_column,
pattern))
        results.extend(batch_results)

        # 顯示進度
        if (start_idx // batch_size) % 10 == 0:
            print(f"已處理 {start_idx + len(batch)}/{len(df)} 行
({((start_idx + len(batch))/len(df)*100):.1f}%)")

    return pd.Series(results, index=df.index)


stop_words = set(stopwords.words('english'))

def tokenize_and_remove_stopwords(text):
    words = word_tokenize(text)
    return [word for word in words if word not in stop_words]


def preprocess_batch(texts, stop_words_set, batch_size=1000):
    """
    批次處理文本的函數

    Parameters:
        texts: 要處理的文本列表
        stop_words_set: 停用詞集合
        batch_size: 批次大小
    """
    results = []
    total = len(texts)

    for i in range(0, total, batch_size):
        batch = texts[i:min(i + batch_size, total)]
```

```
        batch_results = []

        for text in batch:
            if pd.isna(text):
                batch_results.append([])
                continue

            # 使用快速的列表解析
            words = word_tokenize(str(text))
            cleaned = [word for word in words if word not in stop_words_set]
            batch_results.append(cleaned)

        results.extend(batch_results)

        # 顯示進度
        if (i // batch_size) % 10 == 0:
            print(f"已處理 {i + len(batch)}/{total} 行 ({((i +
len(batch))/total*100):.1f}%)")

    return results

def fast_text_cleaning(df, text_column='text', batch_size=1000,
n_jobs=None):
    """
    快速的文本清理函數

    Parameters:
        df: 輸入的 DataFrame
        text_column: 文本列的名稱
        batch_size: 每個批次處理的行數
        n_jobs: 並行處理的作業數，None 表示使用所有可用的 CPU 核心
    """
    # 將停用詞轉換為集合以加快查詢速度
    stop_words_set = set(stop_words)

    if n_jobs and n_jobs > 1:
        # 分割數據
        splits = np.array_split(df[text_column], n_jobs)

        # 創建部分函數，固定其他參數
        process_func = partial(preprocess_batch,
stop_words_set=stop_words_set, batch_size=batch_size)

        # 並行處理
        with ProcessPoolExecutor(max_workers=n_jobs) as executor:
            results = []
            for batch_result in executor.map(process_func, splits):
                results.extend(batch_result)
```

```
        return pd.Series(results, index=df.index)
    else:
        # 單進程處理
        return pd.Series(
            preprocess_batch(df[text_column].values, stop_words_set,
batch_size),
            index=df.index
        )
```

2. lemmatize

```
import spacy
import pandas as pd
import numpy as np
from concurrent.futures import ProcessPoolExecutor
from tqdm import tqdm

def batch_lemmatize(batch_tokens, nlp):
    """
    批次處理詞形還原
    """
    results = []
    # 使用 pipe 來批次處理文本
    texts = [" ".join(tokens) if isinstance(tokens, list) else "" for tokens
in batch_tokens]

    # 使用 spacy 的 pipe 功能進行批次處理，移除 n_process 參數
    for doc in nlp.pipe(texts, batch_size=1000, disable=["parser", "ner"]):
        lemmas = [token.lemma_ for token in doc]
        results.append(lemmas)

    return results

def fast_lemmatize(df, tokens_column='cleaned_text', batch_size=1000):
    """
    快速的詞形還原函數

    Parameters:
        df: 輸入的 DataFrame
        tokens_column: 包含標記的列名
        batch_size: 每批處理的數量
    """
    # 載入 spacy 模型，並禁用不必要的組件
    nlp = spacy.load("en_core_web_sm", disable=["parser", "ner"])

    # 初始化結果列表
    results = []
```

```python
    # 計算總批次數
    total_batches = (len(df) + batch_size - 1) // batch_size

    # 使用 tqdm 顯示進度條
    for i in tqdm(range(0, len(df), batch_size), total=total_batches,
desc="Processing"):
        # 取得當前批次
        batch = df[tokens_column].iloc[i:i + batch_size]

        # 處理當前批次
        batch_results = batch_lemmatize(batch.values, nlp)
        results.extend(batch_results)

    return pd.Series(results, index=df.index)

def process_in_parallel(df, tokens_column='cleaned_text', n_partitions=2):
    """
    Parameters:
        df: 輸入的 DataFrame
        tokens_column: 包含標記的列名
        n_partitions: 分割數量 (建議在 Kaggle 環境使用 2)
    """
    # 分割數據
    dfs = np.array_split(df, n_partitions)

    # 並行處理每個分割
    with ProcessPoolExecutor(max_workers=n_partitions) as executor:
        futures = []
        for sub_df in dfs:
            future = executor.submit(fast_lemmatize, sub_df, tokens_column)
            futures.append(future)

        # 收集結果
        results = []
        for future in futures:
            results.append(future.result())

    # 合併結果
    return pd.concat(results)

# 如果想要使用並行處理，可以這樣調用：
tweets_DM_new['lemmatized_cleaned_text'] = process_in_parallel(
    tweets_DM_new,
    tokens_column='cleaned_text',
    n_partitions=2  # 在 Kaggle 上使用 2 個分割
)
```

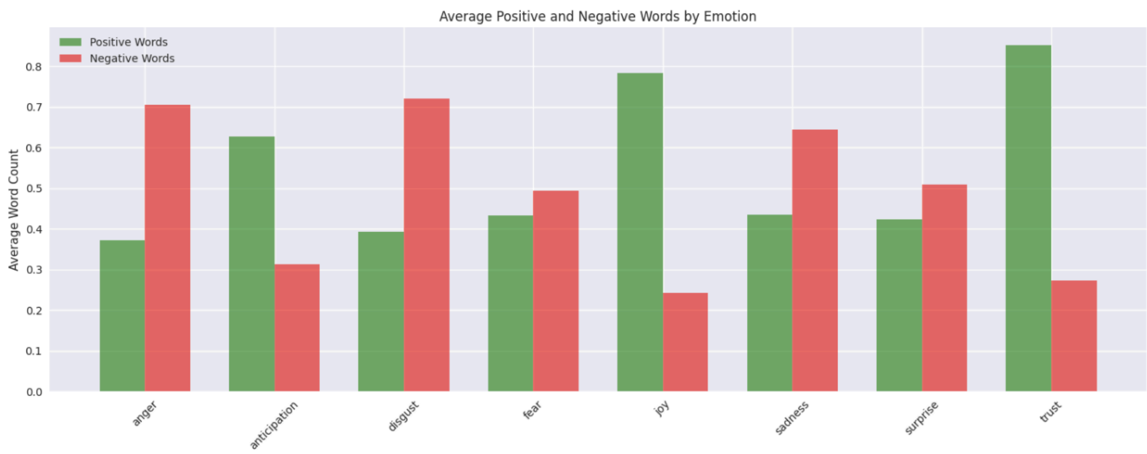
3. Feature Engineering

```python
def process_batch(batch_data, positive_words, negative_words):
    """
    處理一批數據的函數
    """
    pos_counts = []
    neg_counts = []

    pos_set = set(positive_words)
    neg_set = set(negative_words)

    for words in batch_data:
        if not isinstance(words, list):
            pos_counts.append(0)
            neg_counts.append(0)
            continue

        word_set = set(words)
        pos_count = len(word_set.intersection(pos_set))
        neg_count = len(word_set.intersection(neg_set))

        pos_counts.append(pos_count)
        neg_counts.append(neg_count)

    return pos_counts, neg_counts

def create_features_fast(df, column='cleaned_text', batch_size=10000,
n_jobs=2):
    """
    Parameters:
        df: DataFrame 包含文本數據
        column: 包含清理後文本的列名
        batch_size: 每批處理的數據量
        n_jobs: 並行處理的作業數量
    """
    # 載入情感詞典
    positive_words = set(opinion_lexicon.positive())
    negative_words = set(opinion_lexicon.negative())

    # 將數據分割成批次
    n_batches = (len(df) + batch_size - 1) // batch_size
    pos_counts_all = []
    neg_counts_all = []

    # 使用進程池進行並行處理
    with ProcessPoolExecutor(max_workers=n_jobs) as executor:
        futures = []
```

```
        # 提交所有批次的任務
        for i in range(n_batches):
            start_idx = i * batch_size
            end_idx = min((i + 1) * batch_size, len(df))
            batch = df[column].iloc[start_idx:end_idx]

            future = executor.submit(process_batch, batch, positive_words,
negative_words)
            futures.append(future)

            # 顯示進度
            if i % 10 == 0:
                print(f"Submitted batch {i+1}/{n_batches}")

        # 收集結果
        for i, future in enumerate(futures):
            pos_counts, neg_counts = future.result()
            pos_counts_all.extend(pos_counts)
            neg_counts_all.extend(neg_counts)

            if i % 10 == 0:
                print(f"Processed batch {i+1}/{n_batches}")

    # 將結果添加到DataFrame
    df['positive_word_count'] = pos_counts_all
    df['negative_word_count'] = neg_counts_all

    return df
```

```
tweets_DM_new['positive_more'] = tweets_DM_new['positive_word_count'] -
tweets_DM_new['negative_word_count']
```

the result of feature engineering：



Statistical Summary by Emotion:

	positive_word_count			negative_word_count		\
	mean	std	count	mean	std	
emotion						
anger	0.37	0.63	39867	0.70	0.92	
anticipation	0.63	0.86	248935	0.31	0.65	
disgust	0.39	0.64	139101	0.72	0.90	
fear	0.43	0.67	63999	0.49	0.76	
joy	0.78	0.92	516017	0.24	0.55	
sadness	0.43	0.68	193437	0.64	0.87	
surprise	0.42	0.69	48729	0.51	0.75	
trust	0.85	0.94	205478	0.27	0.60	

	count
emotion	
anger	39867
anticipation	248935
disgust	139101
fear	63999
joy	516017
sadness	193437
surprise	48729
trust	205478

Correlation between Positive and Negative Words by Emotion:

anticipation: 0.03

sadness: 0.017

fear: 0.022

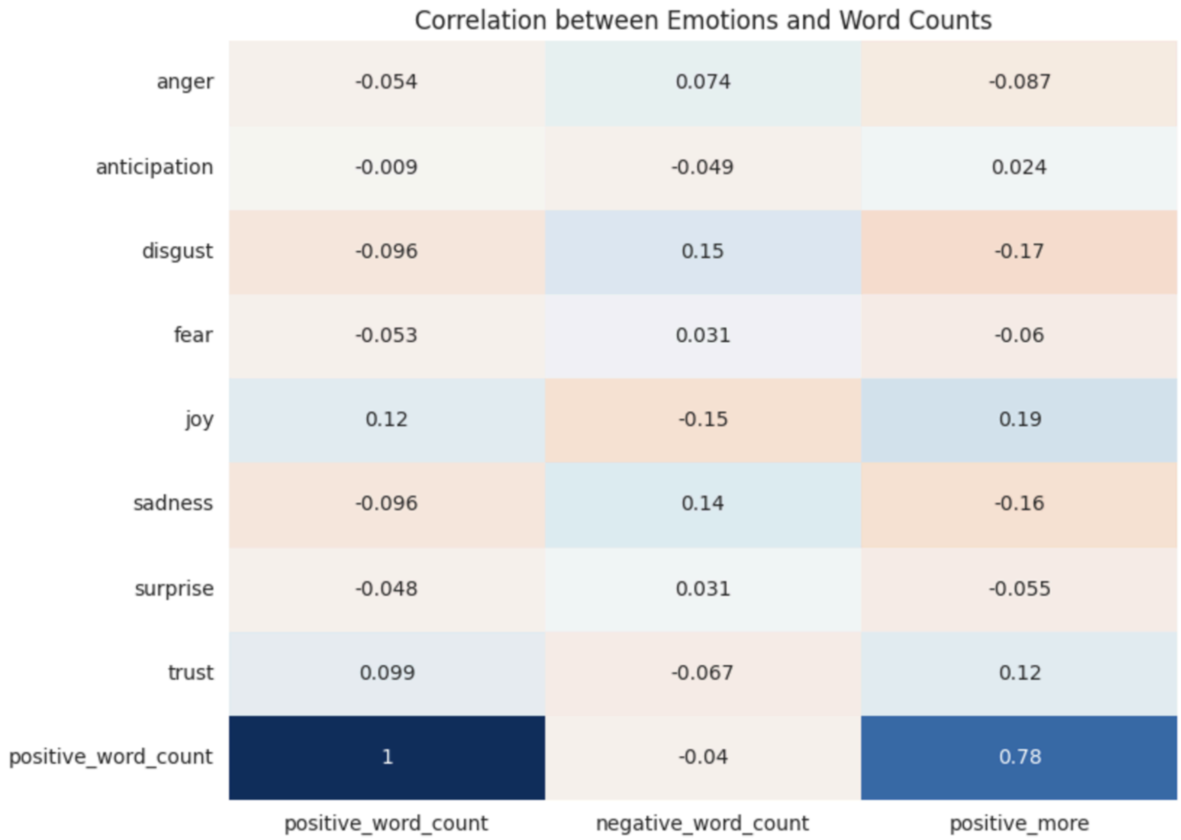
joy: 0.003

anger: 0.039

trust: -0.023

disgust: 0.019

surprise: 0.001



Looking at the results, we can see that there were almost no significant positive or negative correlations.

Since I couldn't think of additional feature engineering methods,I decided to use a pre-trained model as an alternative approach.

Method 2：

data preprocessing

```
def clean_at(text):
    text = re.sub(r'@\w+', '', text)
    return text

def to_lower(text):
    text = text.lower()
    return text
```

```python
def clean_LH(text):
    text = text.replace("<LH>", '')
    return text

def standard_space(text):
    text = re.sub(r'\s+',' ', text)
    return text

def remove_url(text):
    # 使用正則表達式匹配 www. 開頭和 .com 結尾的網址
    # 這個正則表達式會匹配以 "www." 開頭且以 ".com" 結尾的網址
    text = re.sub(r'http[s]?://(?:www\.)?[\w-]+\.[a-z]{2,}/?', '', text)
    return text

def handle_hashtag(text, hashtags):
    if "#" in text:
        found_hashtags = re.findall(r'#\w+', text)

        for hashtag in found_hashtags:
            clean_hashtag = hashtag.lstrip('#')

            if clean_hashtag not in hashtags:
                hashtags.append(clean_hashtag)

    return hashtags

# 將處理函數移到最外層
def process_single_row(args):
    """
    處理單行數據的函數
    """
    row, text_column, hashtags_column, pattern = args
    text = str(row[text_column])
    current_hashtags = row[hashtags_column]

    # 確保 current_hashtags 是列表
    if not isinstance(current_hashtags, list):
        current_hashtags = []

    if '#' in text:
        # 找出所有 hashtags
        found_tags = pattern.findall(text)
        # 移除 # 符號並轉換為集合
        new_tags = {tag.lstrip('#') for tag in found_tags}
        # 現有的 hashtags 轉換為集合
        current_tags = set(current_hashtags)
```

```
        # 合併並轉回列表
        return list(current_tags.union(new_tags))

    return current_hashtags

def process_chunk(args):
    """
    處理數據塊的函數
    """
    chunk, text_column, hashtags_column, pattern = args
    result = []
    for _, row in chunk.iterrows():
        result.append(process_single_row((row, text_column,
hashtags_column, pattern)))
    return result

def process_hashtags_fast_v2(df, text_column='text',
hashtags_column='hashtags', batch_size=10000):
    """
    快速處理大量推文中的 hashtags，使用批次處理而不是多進程

    Parameters:
        df: 輸入的 DataFrame
        text_column: 包含文本的列名
        hashtags_column: 包含現有 hashtags 的列名
        batch_size: 每批處理的行數

    Returns:
        處理後的 hashtags Series
    """
    # 預編譯正則表達式
    pattern = re.compile(r'#\w+')

    # 初始化結果列表
    results = []

    # 批次處理數據
    for start_idx in range(0, len(df), batch_size):
        end_idx = min(start_idx + batch_size, len(df))
        batch = df.iloc[start_idx:end_idx]

        # 處理當前批次
        batch_results = process_chunk((batch, text_column,
hashtags_column, pattern))
        results.extend(batch_results)
```

```python
        # 顯示進度
        if (start_idx // batch_size) % 10 == 0:
            print(f"已處理 {start_idx + len(batch)}/{len(df)} 行
({((start_idx + len(batch))/len(df)*100):.1f}%)")

    return pd.Series(results, index=df.index)


stop_words = set(stopwords.words('english'))

def tokenize_and_remove_stopwords(text):
    words = word_tokenize(text)
    return [word for word in words if word not in stop_words]


def preprocess_batch(texts, stop_words_set, batch_size=1000):
    """
    批次處理文本的函數

    Parameters:
        texts: 要處理的文本列表
        stop_words_set: 停用詞集合
        batch_size: 批次大小
    """
    results = []
    total = len(texts)

    for i in range(0, total, batch_size):
        batch = texts[i:min(i + batch_size, total)]
        batch_results = []

        for text in batch:
            if pd.isna(text):
                batch_results.append([])
                continue

            # 使用快速的列表解析
            words = word_tokenize(str(text))
            cleaned = [word for word in words if word not in
stop_words_set]
            batch_results.append(cleaned)

        results.extend(batch_results)

        # 顯示進度
        if (i // batch_size) % 10 == 0:
```

```
            print(f"已處理 {i + len(batch)}/{total} 行 ({((i + 
len(batch))/total*100):.1f}%)")

    return results

def fast_text_cleaning(df, text_column='text', batch_size=1000, 
n_jobs=None):
    """
    快速的文本清理函數

    Parameters:
        df: 輸入的 DataFrame
        text_column: 文本列的名稱
        batch_size: 每個批次處理的行數
        n_jobs: 並行處理的作業數，None 表示使用所有可用的 CPU 核心
    """
    # 將停用詞轉換為集合以加快查詢速度
    stop_words_set = set(stop_words)

    if n_jobs and n_jobs > 1:
        splits = np.array_split(df[text_column], n_jobs)

        process_func = partial(preprocess_batch, 
stop_words_set=stop_words_set, batch_size=batch_size)

        with ProcessPoolExecutor(max_workers=n_jobs) as executor:
            results = []
            for batch_result in executor.map(process_func, splits):
                results.extend(batch_result)

        return pd.Series(results, index=df.index)
    else:
        return pd.Series(
            preprocess_batch(df[text_column].values, stop_words_set, 
batch_size),
            index=df.index
        )
```

In this method, I still use the same way to clean data, including clean_at, remove url, clean_LH and standard_space

model

```
import pandas as pd
import numpy as np
import torch
```

```python
from transformers import AutoModelForSequenceClassification, AutoTokenizer
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import re
from tqdm import tqdm

class TwitterDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def __getitem__(self, idx):
        text = str(self.texts[idx])
        label = self.labels[idx]

        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            padding='max_length',
            truncation=True,
            return_attention_mask=True,
            return_tensors='pt'
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)
        }

def preprocess_text(text):
    # I detect there still exist some <LH>, so I remove it again
    text = re.sub(r'<LH>', '', text)  # 移除 <LH> 標記
    text = text.strip()
    return text

def train_epoch(model, data_loader, optimizer, device):
    model.train()
    total_loss = 0

    for batch in tqdm(data_loader, desc="Training"):
        optimizer.zero_grad()

        input_ids = batch['input_ids'].to(device)
```

```python
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels
        )

        loss = outputs.loss
        total_loss += loss.item()

        loss.backward()
        optimizer.step()

    return total_loss / len(data_loader)

def evaluate_model(model, data_loader, device):
    model.eval()
    predictions = []
    actual_labels = []

    with torch.no_grad():
        for batch in tqdm(data_loader, desc="Evaluating"):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels']

            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask
            )

            _, preds = torch.max(outputs.logits, dim=1)
            predictions.extend(preds.cpu().tolist())
            actual_labels.extend(labels.cpu().tolist())

    return predictions, actual_labels

def main():
    # set cuda
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")

    # load data
    df = pd.read_pickle('tweets_DM_new_for_BERTtweet.pkl')

    # preprocessing data again
    df['processed_text'] = df['text'].apply(preprocess_text)

    # split train and test
```

```python
    train_df = df[df['identification'] == 'train'].copy()
    test_df = df[df['identification'] == 'test'].copy()

    # create label
    label_dict = {label: idx for idx, label in
enumerate(train_df['emotion'].unique())}
    train_df['label'] = train_df['emotion'].map(label_dict)

    # import BERTweet
    tokenizer = AutoTokenizer.from_pretrained('vinai/bertweet-base')
    model = AutoModelForSequenceClassification.from_pretrained(
        'vinai/bertweet-base',
        num_labels=len(label_dict)
    ).to(device)

    # create TwitterDataset
    train_dataset = TwitterDataset(
        texts=train_df['processed_text'].values,
        labels=train_df['label'].values,
        tokenizer=tokenizer
    )

    # create DataLoader
    train_loader = DataLoader(
        train_dataset,
        batch_size=32,
        shuffle=True
    )

    # set lr=0.00002 and train for three epochs
    optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5)
    num_epochs = 3

    for epoch in range(num_epochs):
        print(f"\nEpoch {epoch + 1}/{num_epochs}")
        train_loss = train_epoch(model, train_loader, optimizer, device)
        print(f"Average training loss: {train_loss:.4f}")

    # do prediction on test_df
    test_dataset = TwitterDataset(
        texts=test_df['processed_text'].values,
        labels=[0] * len(test_df),
        tokenizer=tokenizer
    )
    test_loader = DataLoader(test_dataset, batch_size=32)

    predictions, _ = evaluate_model(model, test_loader, device)

    # map numerical label to original label
    reverse_label_dict = {v: k for k, v in label_dict.items()}
    predicted_emotions = [reverse_label_dict[pred] for pred in predictions]
```

```
    test_df['predicted_emotion'] = predicted_emotions

    # output some example
    print("\n預測範例:")
    for text, pred in zip(test_df['text'].head(),
test_df['predicted_emotion'].head()):
        print(f"\n原始文本: {text}")
        print(f"預測情緒: {pred}")

    # save to csv
    test_df.to_csv('predictions.csv', index=False)

    print("\n預測結果已保存到 predictions.csv")

if __name__ == "__main__":
    main()
```

Merge prediction result and tweetID

```
import pandas as pd
import json

import ast

# Function to safely parse the '_source' column and extract 'tweet_id'
def extract_tweet_id(source_str):
    try:
        # Convert string to dictionary
        source_dict = ast.literal_eval(source_str)
        # Navigate to 'tweet_id' within the parsed dictionary
        return source_dict.get('tweet', {}).get('tweet_id', None)
    except (ValueError, SyntaxError):
        return None


predicted = pd.read_csv('predictions.csv')
print(predicted.columns)
# Apply the function to extract 'tweet_id' and create a new 'id' column
predicted['id'] = predicted['_source'].apply(lambda x:  extract_tweet_id(x))

new_predicted = predicted[['id', 'predicted_emotion']]

new_predicted.rename(columns = {"predicted_emotion" : "emotion"}, inplace =
True)

print(new_predicted.head(5))

new_predicted.to_csv('prediction1127.csv', index = False)
```

In the training phase, I train for 3 epochs, each of them spent around 55 minutes to operate.

And the final result of public leaderboard is 0.5673, the private leaderboard is 0.55308.