# DMLab2 report

113062571 CS 張原鳴

# Data Preparation

Initial data: Each row data corresponds to one tweet id

- data identification.csv : Classify the data into train and test data
- emotion.csv : Label the data in different kind of emotion
- tweets_DM.json : Include _score, _index, _source, _crawldata and _type,  _source include:
    - tweet_id
    - hashtag
    - text

# Data Preparation

I extract the csv files into train_data.csv and test_data.csv in data_create.py.

- Feature selection:
  - id
  - text
  - hashtag
  - _score
- I focus on only text part during the training in the end.

```python
import pandas as pd
df = pd.read_json('tweets_DM.json', lines=True)
#print(df)
data_identification = pd.read_csv('data_identification.csv')
data_emotion = pd.read_csv('emotion.csv')
#print(data_identification['identification'].value_counts())
#print(data_identification)
src = df['_source']
df_src = pd.DataFrame([{
    'hashtags': item['tweet']['hashtags'],
    'tweet_id': item['tweet']['tweet_id'],
    'text': item['tweet']['text']
} for item in src])
tweet_df = pd.concat([df_src, df['_score']], axis=1)
#print(df_merged)
#print(df_merged['identification'].value_counts())
df_result = pd.merge(tweet_df, data_identification, on='tweet_id')
#print(df_result)
df_train = df_result[df_result['identification'] == 'train']
df_test = df_result[df_result['identification'] == 'test']
df_train = pd.merge(df_train, data_emotion, on='tweet_id')

df_train = df_train[['tweet_id', 'text', 'hashtags', '_score', 'emotion']]
df_test = df_test[['tweet_id', 'text', 'hashtags', '_score']]

df_train.to_csv('train_data.csv', index=False)
df_test.to_csv('test_data.csv', index=False)

print(df_train)
print(df_test)
```

# Data Preprocessing

- TfidfVectorizer:
    - Transform a collection of text data in train data into a numerical matrix based on the TF-IDF (Term Frequency-Inverse Document Frequency) score.
    - Limits the vocabulary size to the top 25,000 most important words across all documents, ranked by their TF-IDF score.
- Stop_word = 'english':
    - Automatically removes common English stop words (e.g., "the", "is", "and") from the text before computing TF-IDF,  ignoring words that are meaningless.

In [5]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
TFIDF = TfidfVectorizer(max_features=25000, stop_words='english')
TFIDF.fit(train_df['text'])
```

Out[5]:
```
TfidfVectorizer
TfidfVectorizer(max_features=25000, stop_words='english')
```

# Data Preprocessing

- Split and transform the training data into train and validation part (8:2).
  - We can observe the results not only in training phase but also in validation phase.
- LabelEncoder:
  - Models cannot process text labels (e.g., “anticipation”) directly and need numerical inputs, so I use LabelEncoder for one-hot encoding.
  - Dimension of y_train: 1 to 8 (number of target class).

In [7]:
```
X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42
)
```

In [8]:
```
label_encoder = LabelEncoder()
label_encoder.fit(y_train)
```

Out[8]:
```
▾ LabelEncoder
LabelEncoder()
```

# Create the Model

- Model: Deep neural network (DNN) with six hidden layers
- Regularization:
  - Batch Normalization ensures stable and faster training.
  - Dropout reduces overfitting and prevents the network from becoming overly reliant on specific neurons.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer (InputLayer) | (None, 25000) | 0 |
| dense (Dense) | (None, 1024) | 25,601,024 |
| batch_normalization (BatchNormalization) | (None, 1024) | 4,096 |
| re_lu (ReLU) | (None, 1024) | 0 |
| dropout (Dropout) | (None, 1024) | 0 |
| dense_1 (Dense) | (None, 512) | 524,800 |
| batch_normalization_1 (BatchNormalization) | (None, 512) | 2,048 |
| re_lu_1 (ReLU) | (None, 512) | 0 |
| dropout_1 (Dropout) | (None, 512) | 0 |
| dense_2 (Dense) | (None, 256) | 131,328 |
| batch_normalization_2 (BatchNormalization) | (None, 256) | 1,024 |
| re_lu_2 (ReLU) | (None, 256) | 0 |
| dropout_2 (Dropout) | (None, 256) | 0 |
| dense_3 (Dense) | (None, 128) | 32,896 |
| batch_normalization_3 (BatchNormalization) | (None, 128) | 512 |
| re_lu_3 (ReLU) | (None, 128) | 0 |
| dense_4 (Dense) | (None, 64) | 8,256 |
| batch_normalization_4 (BatchNormalization) | (None, 64) | 256 |
| re_lu_4 (ReLU) | (None, 64) | 0 |
| dense_5 (Dense) | (None, 32) | 2,080 |
| batch_normalization_5 (BatchNormalization) | (None, 32) | 128 |
| re_lu_5 (ReLU) | (None, 32) | 0 |
| dense_6 (Dense) | (None, 8) | 264 |

# Training the Model

- Epoch: 30, batch size: 256
- Early_Stopping: Stop the training process when the model's performance on a validation dataset stops improving.
- Generate the output  result into submission.csv

```python
epochs = 30
batch_size = 256

from tensorflow.keras.callbacks import EarlyStopping

# Define EarlyStopping
early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=2,
    mode='max',
    restore_best_weights=True
)

# training!
history = model.fit(
    X_train, y_train,
    epochs=epochs,
    batch_size=batch_size,
    validation_data=(X_val, y_val),
    callbacks=[early_stopping]
)
```

# Trying for Improvement

I try some methods, but the score just dropped, so I put in here.

- Sampling partial data based on frac
- Process data balancing.

Counts of every class in training data:

```
emotion
joy             516017
anticipation    248935
trust           205478
sadness         193437
disgust         139101
fear             63999
surprise         48729
anger            39867
Name: count, dtype: int64
```

```
emotion
surprise        516017
disgust         516017
trust           516017
anger           516017
fear            516017
sadness         516017
anticipation    516017
joy             516017
Name: count, dtype: int64
```

Speculation of the worse result:

1. Bad quality of generating data

   because text data is hard to simulate.

2. Overfitting problem.

# Trying for Improvement

- Try different models:
  - Random Forest
  - Transformer embeddings
- Speculation of the worse result:
  - The size of training data is too big,  Random Forests are effective on smaller datasets, they may not scale well with such a large amount of data.
  - Lack of fine-tune in pretrained Transformer model, may not align perfectly with our task.

**Thanks**