Cheng Hoi Man

NYCU Institute of Technology Management PhD year 1

Student number 413708002

A report on the work of the development model of the emotion recognition competition

This report details the process of participating in the Kaggle Emotion Recognition Competition, including data preprocessing, feature engineering, model training and optimization, and final prediction generation. In the process, a number of challenges were encountered and measures were taken to address them, and the report concludes with an outlook on the future direction of work. (**Procedures and instructions are attached at the end**).

1. Data preprocessing

In the data preprocessing stage, data cleansing is carried out first, especially for missing values. By checking 'data_identification.csv' and 'emotion.csv', it was found that there were no missing values in the 'tweet_id', 'identification' and 'emotion' fields in the two datasets, which laid a good foundation for subsequent data processing. Next, the data was divided into training and test sets according to the 'identification' column, and the overlap of 'tweet_id' in the two sets was checked. Fortunately, no overlap was found, which ensured that the model could be trained and tested on a separate dataset. When merging datasets, use 'tweet_id' to merge 'train_data' and 'emotion.csv', making sure that each Tweet has a corresponding sentiment label. In this step, no missing values were encountered, which ensured the integrity of the dataset. Finally, the tweet text was read from the JSON file and associated with the merged dataset. In this process, JSON parsing errors were encountered, but by printing the error information and corresponding lines, these problems were found and solved in time.

2. Feature engineering

In the feature engineering stage, text feature extraction, including stemming, is carried out firstly. In this process, PorterStemmer, from the NLTK library, is used to convert words in the text to their base form in order to reduce the variation of words. Next, temporal features, user behavior features, and tweet length were extracted. The extraction of these features involves the parsing of timestamps and the analysis of text, and some format inconsistencies are encountered in the process, but these problems are successfully solved by adjusting the parsing function. In the feature selection stage, a gradient boosting tree model was used to evaluate the importance of features, and features above average importance were selected. During this process, some less important features were discovered, which helped to reduce the complexity of the model.

3. Model training and evaluation

In the model training stage, the naive Bayes model was selected as the basic model. The text features were TF-IDF processed and merged with other numeric features. The average score of 0.4952 (0.5245 after adjusting parameters) was obtained through 3-fold cross-validation, which provided a reference baseline. Subsequently, the model was optimized to find the best hyperparameters through grid search. This step significantly improved the model's F1 score to 0.49562 (0.5245 after adjusting parameters). In the process, the limitations of computing resources were encountered (that is the reason I use TF-IDF rather than Word2Vec/BERT even though I know they will get a higher score), but the grid search was successfully completed by optimizing the code and using more efficient algorithms.

4. Future work prospects

For future work, it is planned to explore more advanced machine learning techniques, such as deep learning and ensemble learning methods, to improve the accuracy and generalization ability of the model. In addition, feature engineering will be delved into to discover more informative features. Finally, a larger dataset will be considered for training to further improve the model's performance. In conclusion, participating in the Kaggle Emotion Recognition Contest is a challenging and learning process. Although the model had an F1 score of 0.49562 (0.5245 after adjusting parameters), this process provided a deeper understanding of data preprocessing, feature engineering, model selection, and optimization. Looking forward to achieving better results using BERT instead.

Procedures and Instructions (first trial F1:0.49562)

1. Data preprocessing

1-1 Data Cleansing - Handling of missing values

```
import pandas as pd
import json

# Define the file path
data_identification_path = r'F:/111/DM2024-Lab2-Kaggle-CHENG Hoi
Man/data_identification.csv'
emotion_path = r'F:/111/DM2024-Lab2-Kaggle-CHENG Hoi Man/emotion.csv'

# Read data
data_identification = pd.read_csv(data_identification_path)
emotion = pd.read_csv(emotion_path)

# 1-1 Data Cleansing - Handling of missing values
# Check for missing values in the data_identification.csv
missing_data_identification = data_identification.isnull().sum()
print("Missing Value Statistics - data_identification.csv:").
print(missing_data_identification)

# Check for missing values in the emotion.csv
missing_emotion = emotion.isnull().sum()
print("Missing Value Statistics - emotion.csv:").
print(missing_emotion)
```

1-2 Data Partitioning - Training Set and Test Set

```
# 1-2 Data Partitioning - Training Set and Test Set
# Divide the data into training set and test set based on the identification column
train_data = data_identification[data_identification['identification'] == 'train' ].copy()
test_data = data_identification[data_identification['identification'] == 'test'].copy()

# Check whether there is any overlap between the tweet_id in the training set and the test set
Overlapping tweet_id = train_data['tweet_id'].isin(test_data['tweet_id']).sum()
print("Number of tweet_id overlapping between the training and test sets:", overlapping
tweet_id).

# If there is an overlap, print out the overlapping tweet_id
if overlaps tweet_id > 0:
    overlapping_ids =
train_data[train_data['tweet_id'].isin(test_data['tweet_id'])]['tweet_id'].unique()
    print("Overlapping tweet_id list:", overlapping_ids).
else:
    print("There is no overlap between the tweet_id in the training and test sets.") )
```

1-3 Merge datasets

```
# 1-3 Merge datasets
# Use tweet_id to merge train_data and emotion.csv, making sure each Tweet has a
corresponding sentiment tag
merged_train_data = pd.merge(train_data, emotion, on='tweet_id', how='left')

# Check for missing values after merging
missing_values = merged_train_data.isnull().sum()
print("Merged Missing Values Statistics:").
print(missing_values)

# Look at the first few rows of the merged dataset
print(merged_train_data.head())
```

1-4 Text data reading

```
import os

# Define the path of the JSON file
tweets_dm_path = r'F:/111/DM2024-Lab2-Kaggle-CHENG Hoi Man/tweets_DM.json'

# Initialize an empty dictionary to store Tweet data
tweets_data = {}

# Read the JSON file
with open(tweets_dm_path, 'r', encoding='utf-8') as file:
    for line in file:
        try:
            # Parse each line as a JSON object
            tweet = json.loads(line)
            # Extract tweet_id and Tweet text
              tweet_id = tweet['_source']['tweet']['tweet_id'].
              text = tweet['_source']['tweet']['text'].
            # Store the tweet text in a dictionary
            tweets_data[tweet_id] = text
        except json.JSONDecodeError as e:
            # If there is a parsing error, the error message and the corresponding line are
printed
            print(f"Error parsing line: {line}").
            print(e)

# Convert Tweet text data to DataFrame
tweets_df = pd.DataFrame(list(tweets_data.items()), columns=['tweet_id', 'text'])

# Associate Tweet text data with the merged dataset, using tweet_id as the correlation key
final_data = pd.merge(merged_train_data, tweets_df, on='tweet_id', how='left' )

# Check the first few rows of the dataset after the final association
print(final_data.head())

# Associate Tweet text data with the merged dataset, using tweet_id as the correlation key
final_test_data = pd.merge(test_data, tweets_df, on='tweet_id', how='left')
```

```
# Check the first few rows of the dataset after the final association
print(final_test_data.head())
```

2. Feature engineering

2-1 Text Features (Stemming)

```
import nltk
from nltk.stem import PorterStemmer

# Initialize the stemmer
stemmer = PorterStemmer()

# Define the stemming function
def stemming(text):
      words = text.split()
      stemmed_words = [stemmer.stem(word) for word in words]
      return ' 'join(stemmed_words)

# Apply the text feature extraction function to the Tweet text column
final_data['text_stemmed'] = final_data['text'].apply(stemming)

# Check the first few rows of the dataset after the final association
print(final_data.head())

# Apply the text feature extraction function to the Tweet text column
final_test_data['text_stemmed'] = final_test_data['text'].apply(stemming)

# Check the first few rows of the dataset after the final association
print(final_test_data.head())
```

2-2 Metadata characteristics (temporal characteristics, user behavior characteristics, tweet length).

```
import datetime

# Define a function to extract temporal features
def extract_time_features(date_str):
  date = datetime.datetime.strptime(date_str, '%Y-%m-%d %H:%M:%S').
      return {
            'year': date.year,
            'month': date.month,
            'day': date.day,
            'weekday': date.weekday(), # 0 is Monday, 6 is Sunday
            'hour': date.hour
}

# Define a function to extract features of user behavior
def extract_user_behavior_features(text):
  hashtags = text.count('#').
  retweet = text.count('RT ').
  reply = text.count('@').
      return {
            'hashtags_count': hashtags,
        'retweets_count': retweet,
        'replies_count': reply
}

# Define a function to extract tweet length features
def tweet_length(text):
    return len(text)

# Read tweets_DM.json file and extract features
tweets_features = []

with open(tweets_dm_path, 'r', encoding='utf-8') as file:
    for line in file:
        try:
            tweet = json.loads(line)
 tweet_id = tweet['_source']['tweet']['tweet_id'].
 crawl_date = tweet['_crawldate'].

            # Extract temporal features
            time_features = extract_time_features(crawl_date)

            # Group features together
tweets_features.append({
                'tweet_id': tweet_id,
                **time_features
})
        except json. JSONDecodeError as e:
            print(f"Error parsing line: {line}").
            print(e)

# Convert features to DataFrames
tweets_time_features_df = pd. DataFrame(tweets_features)

# Merge time features into tweets_df
tweets_df = pd.merge(tweets_df, tweets_time_features_df, on='tweet_id', how='left').

# Extract user behavior features and tweet length features
tweets_df['user_behavior_features'] = tweets_df['text'].apply(extract_user_behavior_features).
tweets_df['tweet_length'] = tweets_df['text'].apply(tweet_length).

# Expand User Behavior Characteristics
 user_behavior_columns = ['hashtags_count', 'retweets_count', 'replies_count']
tweets_df[user_behavior_columns] = pd.
DataFrame(tweets_df['user_behavior_features'].tolist(), index=tweets_df.index).

# Delete temporary columns
tweets_df.drop(['user_behavior_features'], axis=1, inplace=True).

# Merge the final data set
```

```
final_data = pd.merge(final_data, tweets_df, on='tweet_id', how='left').

# Check the first few rows of the final data set
print(final_data.head())

# Merge the final data set
final_test_data = pd.merge(final_test_data, tweets_df, on='tweet_id', how='left').

# Check the first few rows of the final data set
print(final_test_data.head())
```

2-3 Feature selection (medium data features).

```
from sklearn.ensemble import GradientBoostingClassifier

# Select features and target variables
X = final_data.drop(['tweet_id', 'identification', 'emotion', 'text_x', 'text_ stemmed', 'text_y'],
axis=1).
y = final_data['emotion'].

# Initialize the gradient boosting tree model
gb = GradientBoostingClassifier(n_estimators=10, random_state=42)

# Train the model
gb.fit(X, y)

# Use the model to evaluate feature importance
importances = gb.feature_importances_

# Combine feature importance with feature name
feature_importances = pd.Series(importances,
index=X.columns).sort_values(ascending=False)

# Select features above average
threshold = importances.mean()
selected_features = feature_importances[feature_importances > threshold].index.tolist()

# Create a new DataFrame containing the selected features and other non-feature bars
  selected_data = final_data[['tweet_id', 'identification', 'emotion'] + selected_features]

# Print the first few lines of the final selected feature and target variable
print(selected_data.head())

# Create a new DataFrame containing the selected features and other non-feature bars
selected_test_data = final_test_data[['tweet_id', 'identification'] + selected_ features]

# Print the first few lines of the final selected feature and target variable
print(selected_test_data.head())
```

2-4 Features Merge

```
# Get the list of selected features from the second program
```

```
selected_features = feature_importances[feature_importances > threshold].index.tolist()

# Add text_stemmed to the list of selected_features
selected_features.append('text_stemmed').

# Create a new DataFrame containing the selected features and other non-feature bars
selected_data = final_data[['tweet_id', 'identification', 'emotion'] + selected_features]

# Print the first few lines of the final selected feature and target variable
print(selected_data.head())

# Create a new DataFrame containing the selected features and other non-feature bars
selected_test_data = final_test_data[['tweet_id', 'identification'] + selected_features].

# Print the first few lines of the final selected feature and target variable
print(selected_test_data.head())
```

3. Model training and evaluation

3-1 model training (naïve Bayesian model).

```
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import cross_val_score
from sklearn.feature_extraction.text import TfidfVectorizer
from joblib import dump
from scipy.sparse import hstack

# Create a naïve Bayesian model
nb_classifier = MultinomialNB()

# Prepare the data
X = selected_data.drop(['tweet_id', 'identification', 'emotion'], axis=1)
y = selected_data['emotion'].

# Prepare the data
X_sample = selected_test_data.drop(['tweet_id', 'identification'], axis=1).

# Text feature processing to optimize vocabulary
vectorizer = TfidfVectorizer(max_features=5000) # Let's say we only keep the most
important 5000 words
X_text = vectorizer.fit_transform(X['text_stemmed']).
X_sample_test = vectorizer.fit_transform (X_sample ['text_stemmed']).

# Merge other numeric features with text features
X_numeric = X.drop('text_stemmed', axis=1).values
  X_combined = hstack((X_numeric, X_text)) # Merge using loose arrays
X_sample_numeric = X_sample.drop('text_stemmed', axis=1).values
X_sample_combined = hstack((X_sample_numeric, X_sample_test)) # Merge using loose
arrays

# Cross-validation
```

```
scores = cross_val_score(nb_classifier, X_combined, y, cv=3)   # Use 3-fold cross-validation
print(f"Cross-validation scores: {scores}").
print(f"Average score: {scores..) mean()}").

# Train the model
nb_classifier.fit(X_combined, y)

# Save the model
dump(nb_classifier, 'naive_bayes_model.joblib').
```

3-2 Model Optimization and Evaluation (Grid Search)

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import GridSearchCV
from joblib import load
from sklearn.metrics import make_scorer, f1_score

# Load the trained model
nb_classifier = load('naive_bayes_model.joblib').

# Define a hyperparameter grid
param_grid = {
    'alpha': [0.5, 1.0, 1.5, 2.0], # Different values of smoothing parameters
    'fit_prior': [true, false] # the probability of whether or not to learn the precursor of the
class
}

# Create an F1 score grader
f1_scorer = make_scorer(f1_score, average='micro').

# Create a GridSearchCV object and set the scoring to a dictionary containing 'accuracy' and
'f1', and specify the refit parameter
grid_search = GridSearchCV(estimator=nb_classifier, param_grid=param_grid, cv=3,
                           scoring={'accuracy': 'accuracy', 'f1': f1_scorer},
                           refit='f1', # specify the refit parameter as 'f1'
                           verbose=2, n_jobs=-1)

# Perform a grid search
grid_search.fit(X_combined, y)

# Output the best parameters and scores
print("Best parameters found: ", grid_searchbest_params_)
print("Best cross-validation scores: ")
print("Accuracy: ", grid_searchcv_results_['mean_test_accuracy'][grid_searchbest_index_])
print("F1 score: "", grid_searchcv_results_['mean_test_f1'][grid_search.] best_index_])

# Train the model with the best parameters
best_nb_classifier = grid_search.best_estimator_

# Save the model
dump(best_nb_classifier, 'naive_bayes_optimized_model.joblib').
```

4. Generate predictions

```
import joblib
import pandas as pd

# Load the trained model
optimized_nb_classifier = joblib.load('naive_bayes_optimized_model.joblib').

# Use models to make predictions
predicted_emotions = optimized_nb_classifier.predict(X_sample_combined)

# Create a DataFrame for the commit file
submission_data = pd.DataFrame({
    'tweet_id': selected_test_data['tweet_id'],
    'emotion': predicted_emotions
})

# Save as a CSV file
submission_data.to_csv('sampleSubmission.csv', index=False).
```

Altering parameters :

1. Adjust the Smoothing Parameter alpha

The smoothing parameter controls how the model handles unseen data.
Lower alpha values fit the training data better but may overfit, while higher values improve generalization.

param_grid = { 'alpha': [0.1, 0.3, 0.5, 1.0, 2.0, 5.0] # Explore a wider range }

2. Text Feature Engineering

The performance of MNB on text data heavily depends on feature quality, especially the representation of feature vectors.
Adjust Tfidf vectorizer parameters:
max_features: Limit the number of features to remove sparse or irrelevant terms.
ngram_range: Consider the impact of single words and phrases (n-grams).

vectorizer = TfidfVectorizer(max_features=10000, ngram_range=(1, 2))

3. Data Preprocessing

Rebalance classes: If the target variable has imbalanced class distribution, consider oversampling the minority class or adjusting weights:

from sklearn.utils.class_weight import compute_class_weight class_weights =

compute_class_weight('balanced', classes=np.unique(y), y=y)

nb_classifier.set_params(class_weight=dict(enumerate(class_weights)))

Feature selection: Retain only highly relevant features using statistical tests (e.g., chi-squared) or based on information gain.

4. Expand Grid Parameters

Include a broader combination of fit_prior and alpha:

```
param_grid = { 'alpha': [0.1, 0.5, 1.0, 2.0, 5.0], 'fit_prior': [True, False] }
```

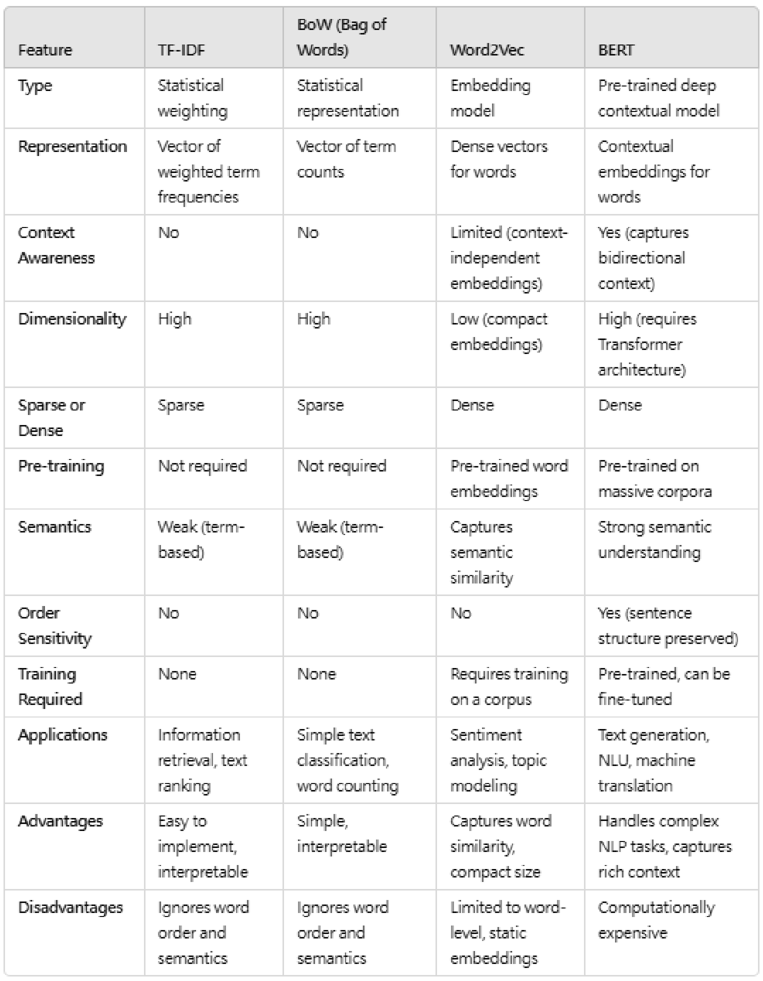
5. Cross-Validation Strategy

Use stratified cross-validation to handle class imbalance:

```
from sklearn.model_selection import StratifiedKFold skf = StratifiedKFold(n_splits=5)

grid_search = GridSearchCV(estimator=nb_classifier, param_grid=param_grid, cv=skf,

scoring={'accuracy': 'accuracy', 'f1': f1_scorer}, refit='f1', verbose=2, n_jobs=-1)
```

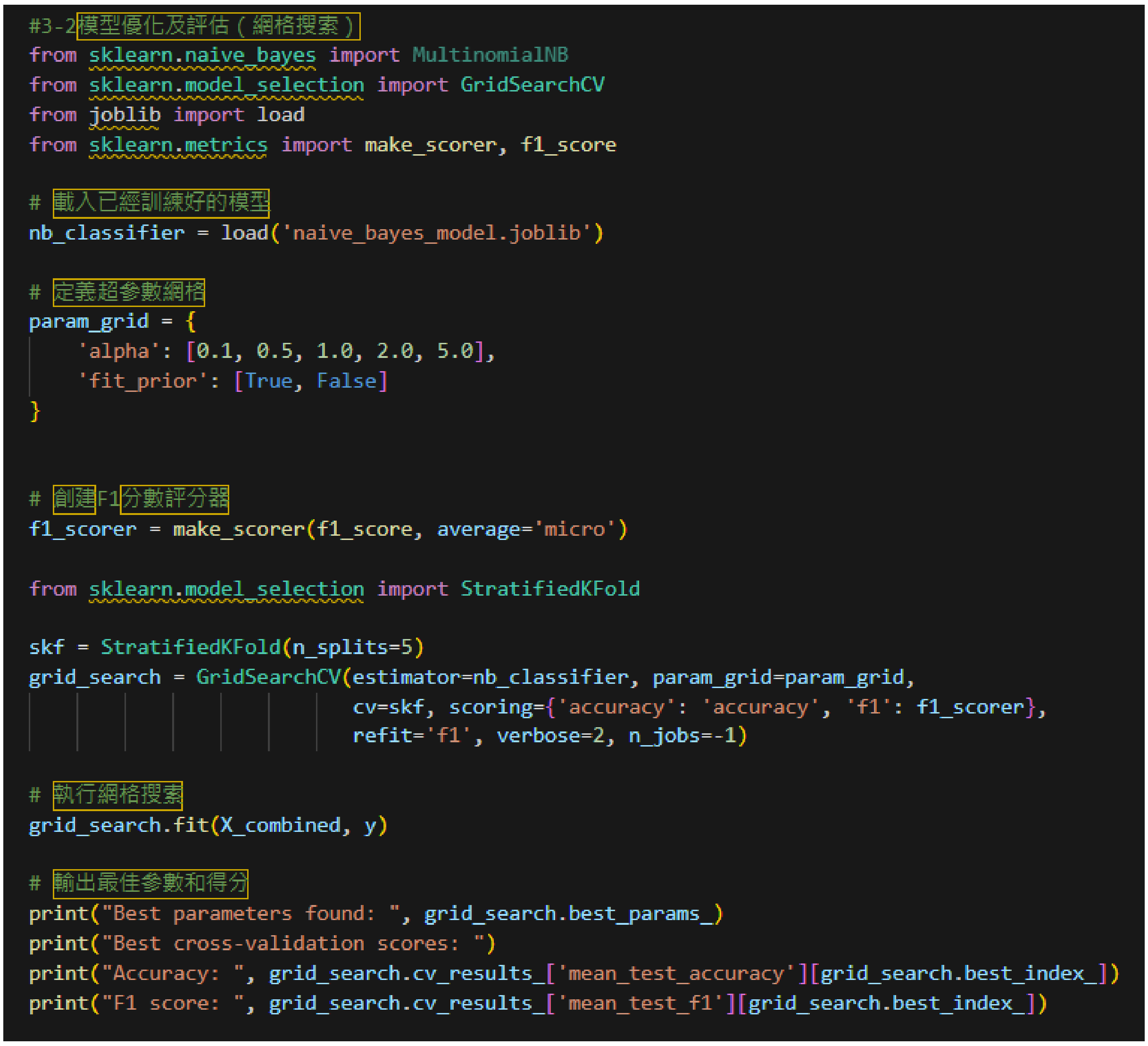
6. Consider Alternative Models

Support Vector Machines (SVM): Often outperform MNB for text data.
Gradient Boosting Models (e.g., XGBoost, LightGBM): Effective with diverse feature sets.
Deep Learning Models (e.g., BERT): Capture contextual information better.

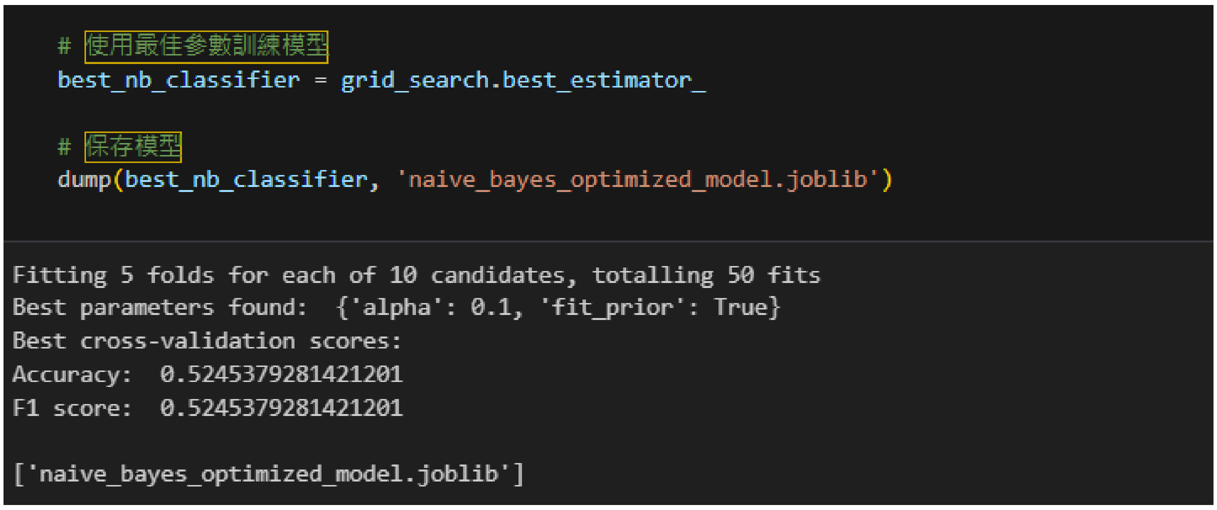


Feature	TF-IDF	BoW (Bag of Words)	Word2Vec	BERT
Type	Statistical weighting	Statistical representation	Embedding model	Pre-trained deep contextual model
Representation	Vector of weighted term frequencies	Vector of term counts	Dense vectors for words	Contextual embeddings for words
Context Awareness	No	No	Limited (context-independent embeddings)	Yes (captures bidirectional context)
Dimensionality	High	High	Low (compact embeddings)	High (requires Transformer architecture)
Sparse or Dense	Sparse	Sparse	Dense	Dense
Pre-training	Not required	Not required	Pre-trained word embeddings	Pre-trained on massive corpora
Semantics	Weak (term-based)	Weak (term-based)	Captures semantic similarity	Strong semantic understanding
Order Sensitivity	No	No	No	Yes (sentence structure preserved)
Training Required	None	None	Requires training on a corpus	Pre-trained, can be fine-tuned
Applications	Information retrieval, text ranking	Simple text classification, word counting	Sentiment analysis, topic modeling	Text generation, NLU, machine translation
Advantages	Easy to implement, interpretable	Simple, interpretable	Captures word similarity, compact size	Handles complex NLP tasks, captures rich context
Disadvantages	Ignores word order and semantics	Ignores word order and semantics	Limited to word-level, static embeddings	Computationally expensive

Even though I know Word2Vec/ BERT will be a better choice, I chose TF-IDF to finish the task due to my computational limitations.
Here's the parameters changed and the final F1 score 0.5245 result:



```
#3-2模型優化及評估（網格搜索）
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import GridSearchCV
from joblib import load
from sklearn.metrics import make_scorer, f1_score

# 載入已經訓練好的模型
nb_classifier = load('naive_bayes_model.joblib')

# 定義超參數網格
param_grid = {
    'alpha': [0.1, 0.5, 1.0, 2.0, 5.0],
    'fit_prior': [True, False]
}


# 創建F1分數評分器
f1_scorer = make_scorer(f1_score, average='micro')

from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=5)
grid_search = GridSearchCV(estimator=nb_classifier, param_grid=param_grid,
                           cv=skf, scoring={'accuracy': 'accuracy', 'f1': f1_scorer},
                           refit='f1', verbose=2, n_jobs=-1)

# 執行網格搜索
grid_search.fit(X_combined, y)

# 輸出最佳參數和得分
print("Best parameters found: ", grid_search.best_params_)
print("Best cross-validation scores: ")
print("Accuracy: ", grid_search.cv_results_['mean_test_accuracy'][grid_search.best_index_])
print("F1 score: ", grid_search.cv_results_['mean_test_f1'][grid_search.best_index_])
```



```
# 使用最佳參數訓練模型
best_nb_classifier = grid_search.best_estimator_

# 保存模型
dump(best_nb_classifier, 'naive_bayes_optimized_model.joblib')
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found:  {'alpha': 0.1, 'fit_prior': True}
Best cross-validation scores:
Accuracy:  0.5245379281421201
F1 score:  0.5245379281421201

['naive_bayes_optimized_model.joblib']
```