Angeline (李佳鈺) 110006216

Lab 2 Homework Report

Note: My local machine had some issues and conflicts with the libraries, so I utilized Kaggle platform to run the code for the Kaggle Competition. As for the MASTER, it is a mixture of using google collab and my local machine (specifically on 10.1-10.3) due to the model requiring more system memory (10.3 GiB) than is available for google collab (3.9 GiB). Switching environments might've resulted in not ideal outputs, as the code takes a lot of hours to run. So I didn't try and experiment a lot of preprocessing methods as I would've liked due to long running time and hardware limitations.

Setting up the notebook

Import necessary libraries

```
import json
import pandas as pd
import numpy as np
import nltk
import matplotlib.pyplot as plt
from collections import Counter
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
```

Load and prepare the data

The data was loaded from a JSON file containing the tweets and other CSV files for emotion and identification labels. The JSON file was parsed to extract relevant fields, such as the tweet text and metadata, they are then merged together with the identification and emotion data.

```
# Read data from JSON file
data_path = '/kaggle/input/dm-2024-isa-5810-lab-2-homework/tweets_DM.json'
with open(data_path, 'r') as file:
    data = [json.loads(line) for line in file]

# Load CSV files
emotion_path = '/kaggle/input/dm-2024-isa-5810-lab-2-homework/emotion.csv'
data_identification_path = '/kaggle/input/dm-2024-isa-5810-lab-2-homework/data_identification.csv'
emotion_df = pd.read_csv(emotion_path)
data_identification_df = pd.read_csv(data_identification_path)
```

```python
# Extract data from nested dictionary and create a DataFrame
df = pd.DataFrame({
    'tweet_id': [item['_source']['tweet']['tweet_id'] for item in data],
    'hashtags': [item['_source']['tweet']['hashtags'] for item in data],
    'text': [item['_source']['tweet']['text'] for item in data]
})
```

```python
# Merge the DataFrame with the identification and emotion data
df = df.merge(data_identification_df, on='tweet_id', how='left')
train_data = df[df['identification'] == 'train']
train_data = train_data.merge(emotion_df, on='tweet_id', how='left')
```

Data Cleaning and Preprocessing

- **Duplicate Removal**: Removed duplicate tweets based on their text content.
- **Sampling**: Reduced the dataset size by sampling 10% of the data due to long running time.
- Then implemented a text preprocessing function that performed the following tasks

Text Preprocessing

- **Contraction Expansion**: Converted contractions (e.g., "can't" → "cannot").
- **Special Character Removal**:
    - Removed URLs, HTML tags, and user mentions.
    - Retained only letters, removing digits and special characters.
- **Spelling Correction**: Leveraged TextBlob for spell correction.
- **Part-of-Speech Filtering**: Extracted adjectives using spaCy to retain sentiment-rich features.

```python
# Remove duplicate tweets based on text content
train_data.drop_duplicates(subset=['text'], keep='first', inplace=True)
```

```python
# Sampling the data to reduce the size
train_data_sample = train_data.sample(frac=0.1, random_state=42)
```

```python
import re
import spacy
from textblob import TextBlob

# Load spaCy model for English
nlp = spacy.load('en_core_web_sm')

def expand_contractions(text):
    # Dictionary of English contractions
    contractions_dict = {"can't": "cannot", "won't": "will not", "n't": " not", "'re": " are", "'s": " is", "'d": " would", "'ll": " wi
                         "'t": " not", "'ve": " have", "'m": " am"}
    contractions_re = re.compile('(%s)' % '|'.join(contractions_dict.keys()))

    def replace(match):
        return contractions_dict[match.group(0)]

    return contractions_re.sub(replace, text)

def remove_special_characters(text, remove_digits=False):
    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)
    # Remove user @ references and '#' from tweet
    text = re.sub(r'\@\w+|\#','', text)
    # Remove HTML tags
    text = re.sub(r'<.*?>', '', text)
    # Remove everything that is not letters
    text = re.sub(r'[^a-zA-Z]', ' ', text)
    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()
    return text


def correct_spellings(text):
    return str(TextBlob(text).correct())

def preprocess_text(row):
    text = row['text'] + ' ' + ' '.join(row['hashtags'])
    text = expand_contractions(text)
    text = text.lower()
    text = remove_special_characters(text, remove_digits=True)
    text = correct_spellings(text)
    doc = nlp(text)
    adjectives = [token.text for token in doc if token.pos_ == 'ADJ']
    return ' '.join(adjectives)
```

```python
# Apply preprocessing to train data
train_data_sample['processed_text'] = train_data_sample.apply(preprocess_text, axis=1)
```

```python
# Remove entries without adjectives as they won't be vuseful for our model
train_data_sample = train_data_sample[train_data_sample['processed_text'] != '']
```

```python
# Prepare features and labels
X_train = train_data_sample['processed_text']
y_train = train_data_sample['emotion']
```

Feature Engineering

Text Vectorization

- Used **TF-IDF Vectorizer** with:
    - A maximum feature size of 1000.
    - English stop words removed.

- Generated feature vectors for model training.

Label Encoding

- Encoded emotion labels into numeric format for compatibility.

```
# Encode the labels to be used in the model
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y_train)
```

```
# Using TF-IDF Vectorizer to convert text data into TF-IDF features
tfidf_vectorizer = TfidfVectorizer(max_features=1000, stop_words='english')
X_tfidf = tfidf_vectorizer.fit_transform(X_train).toarray()

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_tfidf, y_encoded, test_size=0.2, random_state=42, stratify=y_encoded)
```

Model (Random Forest Classifier)

Training and Evaluation

- Split the processed dataset into **training (80%)** and **testing (20%)** subsets.
- Trained the Random Forest model with default hyperparameters.
- Evaluated the model:

```
Accuracy: 0.4167650531286895
Classification Report:
             precision    recall  f1-score   support

           0       0.41      0.09      0.15       462
           1       0.34      0.15      0.21      2514
           2       0.36      0.17      0.23      1693
           3       0.41      0.09      0.15       677
           4       0.43      0.83      0.57      6238
           5       0.41      0.27      0.32      2330
           6       0.29      0.06      0.11       524
           7       0.43      0.18      0.25      2502

    accuracy                           0.42     16940
   macro avg       0.38      0.23      0.25     16940
weighted avg       0.40      0.42      0.36     16940
```

```
# Use a Random Forest Classifier
from sklearn.ensemble import RandomForestClassifier

random_forest_model = RandomForestClassifier(random_state=42)
random_forest_model.fit(X_train, y_train)
```

```
# Prediction and Evaluation
y_pred = random_forest_model.predict(X_test)
```

```
from sklearn.metrics import accuracy_score, classification_report

print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 0.4167650531286895
Classification Report:
               precision    recall  f1-score   support

           0       0.41      0.09      0.15       462
           1       0.34      0.15      0.21      2514
           2       0.36      0.17      0.23      1693
           3       0.41      0.09      0.15       677
           4       0.43      0.83      0.57      6238
           5       0.41      0.27      0.32      2330
           6       0.29      0.06      0.11       524
           7       0.43      0.18      0.25      2502

    accuracy                           0.42     16940
   macro avg       0.38      0.23      0.25     16940
weighted avg       0.40      0.42      0.36     16940
```

Predictions

- Applied preprocessing steps to the test dataset.
- Used the trained model to predict emotions on test data.

```
# Identify and prepare the test dataset
test_data = df[df['identification'] == 'test']
```

```
# Preprocess the text data in the test dataset
test_data.loc[:, 'processed_text'] = test_data.apply(preprocess_text, axis=1)

# Convert test data text to TF-IDF features using the same vectorizer trained on the training data
X_test_data = tfidf_vectorizer.transform(test_data['processed_text']).toarray()

# Predict the emotions on the test dataset using the trained model
y_test_pred = random_forest_model.predict(X_test_data)
```

```
# Convert predicted numeric labels back to original emotion labels
predicted_emotions = label_encoder.inverse_transform(y_test_pred)

# Create a DataFrame for the submission, mapping tweet IDs to predicted emotions
submission = pd.DataFrame({
    'id': test_data['tweet_id'],
    'emotion': predicted_emotions
})
```

```
# Output the submission file
submission.to_csv('/kaggle/working/submission.csv', index=False)
```

Methods Tried

Initially, I experimented with simpler algorithms like Naïve Bayes. However, despite multiple iterations and adjusting the training dataset, the accuracy was consistently unsatisfactory. Given the constraint of using only 10% of the full dataset for efficiency, I considered the need for a more complex approach.

Facing computational limitations on Kaggle, I experimented with neural networks. Yet, the computational demand was impractical, I consistently hit Kaggle's runtime limits without yielding tangible results. This experience led me to select a middle ground—decision trees. This model provided a satisfactory compromise between simplicity and computational efficiency, capable of managing the dataset without extensive hardware resources.

Insights Gains

1. Data Quality Impact

Data cleaning significantly affected the outcomes. Cleaning the data, especially removing noise (like URLs and mentions), and correcting spelling mistakes, significantly helped in reducing model confusion, and enhanced model performance. A potential improvement to the model can be made by handling emojis/emoticons. Tweets often contain emojis which can convey important emotional cues. Mapping these to words or special tokens instead of removing them could provide additional sentiment context to the model.

2. Time is money

The limitations caused by hardware were a huge lesson. The need to balance model complexity with available computing resources was a huge issue for me during this lab. Due to the hardware limitations and training taking too long, I have to try multiple environments (local device, google collab, and kaggle) to ensure the code wont stop running or get interruptions in the middle of training. From my experiences after MASTER notebook, running much more complex models can take way longer compared to simpler models, which is obvious theoritically but doing that in practice is very tiring, since my laptop needs to keep on running the entire time and plugged into a socket, this can get in the way of my other courses so I had to take some compromises and work with what I have. I think this lab has opened my eyes in aspects of time and machine being a huge role in data mining. Knowing how model works and theoritical knowledge is very important, but without real life data mining experience, I woud've been taken aback and unprepared with the commitment training a model would be.